Dynaamisesti määriteltävä mallinnustyökalu

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Tieto- ja viestintätekniikan koulutusohjelma, Riihimäen kampus

Kevät, 2022

Marco Peciola

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintätekniikan koulutusohjelma
Tekijä Marco Peciola
Työn nimi Dynaamisesti määriteltävä mallinnustyökalu
Ohjaaja Petri Kuittinen

Tiivistelmä
Vuosi 2022

TIIVISTELMÄ

Opinnäytetyö syntyi tarpeesta ohjelmalle, joka antaisi käyttäjälleen mahdollisuuden kirjata muistiinpanoja tai tarinoita ja palata näihin tarinoihin tarpeen vaatiessa. Ohjelman ydinajatuksena toimi geneerisyys ja mahdollisuus soveltaa ohjelmaa useaan arkipäivän tarpeeseen, jotta käyttäjäkunta olisi mahdollisimman laaja. Kyseistä ohjelmaa suunniteltaessa toimivat kohdeyleisönä kirjoittajat ja matkailijat, mutta sovellusta ei haluttu rajata vain mainitulle käyttäjäkunnalle. Opinnäytetyön aihe tuli Marko Vehmakselta ja hänen visioitansa käytettiin ohjelman pohjustukseen.

Tavoitteeseen pääsy vaati mahdollisimman geneeristä koodia ja tietokantaa, joka olisi valmis laajenemaan ja mukautumaan tarpeen vaatiessa. Työtä varten käytiin läpi monia eri ohjelmointikieliä, tietorakenteita ja kehyksiä, joista lopulliseen ohjelmaan valikoituivat ne, jotka soveltuivat tarkoitukseen parhaiten. Projektin etenemistä hidastivat lukuisat eri vaihtoehdot ja mahdollisuudet, joilla sovellusta olisi voitu viedä eteenpäin. Työn toteuttaminen vaati suurta määrää aiheeseen tutustumista ja suunnittelu veikin suunniteltua enemmän aikaa.

Työssä käydään läpi web sovelluskehityksessä ilmeneviä haasteita ja tapoja selvittää ongelmat. Työn lopputulos miellytti tilaajaa ja haluttuun tavoitteeseen päästiin vaikeuksista huolimatta. Lopullinen sovellus noudattaa suunniteltua säännöttömyyttä ja yllyttää käyttäjää kokeilemaan rajoja. Paikoitellen työn alussa valitut tekniikat aiheuttivat ongelmia ja vaihtoehtojen hyödyt ja haitat nousevat esille. Tulevaisuutta varten jäi paljon kehitettävää, mutta nämä voidaan nähdä mahdollisuutena palata projektin pariin ja lähinnä esteettisinä parannuksina.

Avainsanat ohjelmistosuunnittelu, epälineaarinen datarakenne, web-ohjelmointi
Sivut 66 sivua ja liitteitä 9 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Program in Information and Communication Technology
Author Marco Peciola
Subject Dynamically definable modelling tool
Supervisors Petri Kuittinen

Abstract
Year 2022

ABSTRACT

This thesis was born from the dire need of a colleague, for a program that lets the user create a nonlinear notepad, which could in theory store a whole imaginary world within. The focus and separating factor compared to other similar programs would be the structure. The nonlinear links would permit the user to easily navigate within the program, which in turn would lead to easier saving and deleting of the data stored within the program. Another important goal for the program was to remove boundaries when creating a new nonlinear record. To reach this, the program would have to be as generic as possible, so users can use it in a variety of ways.

Writing a generic program like this, which would dynamically adjust itself based on user choices, while trying to avoid rules determining what the user can create, is not an easy task and I saw this thesis as a possibility for myself to grow as a programmer. The final selections of techniques used in this thesis were a result of trial and error. There were many cases where after an implementation of logic a better way was found, and the application needed to regress to improve itself.

The result satisfied the client, as the project achieved its goals of giving an experience without borders or rules and encourages the user to push boundaries. The road to completion was not easy and some of the choices that were made at the start of the project backfired. A lot of work remains for the future, but these are seen as an opportunity to linger with the project and can be considered as aesthetic improvements.

Keywords nonlinear, data structure, generic programming, web-application
Pages 66 pages and appendices 9 pages

Termit ja lyhenteet

suomi	englanti	Selitys
HTML	HTML	Standardoitu merkintäkieli internetsivujen kirjoittamiseen.
Tag	Elementti	Auttaa selainta tulkitsemaan näytettävän kohteen rakenteen.
Redux	Redux	JavaScript-kirjasto sovellustilan hallintaan.
SQL	SQL	Kyselykieli relaatiokannan hakuja varten.
Single-page application	Yhden sivun sovellus	Sovellus, jossa sivu päivittyy dynaamisesti vuorovaikutuksen johdosta.
Breadcrumb navigation	Murupolku	Auttaa käyttäjiä ylläpitämään tietoisuutta sijainnistaan.
SQL-injection	SQL-injektio	Hyökkäys, jossa yritetään antaa tietokantapalvelimelle kiellettyjä komentoja.
Cross-site scripting	Sivustojen välinen komentosarja	Yritys kiertää pääsynhallinta, kuten saman alkuperän käyttö.
Cross-site request forgery	Istuntoratsastus	Selaimen halutaan luottavan hyökkääjän komentoihin.
DTL	DTL	Djangon käyttämä mallikieli sivujen näyttämiseen.
Object-relational mapping	Objektisuhdekartoitus	tekniikka yhteensopimattomien järjestelmien muuntoa varten.
Interpreter	Ohjelmointikielen tulkki	Käsittelee ja suorittaa ohjelmointikielen lauseita.
Tech Stack	Teknologiapaketti	Projektissa käytetyt tekniikat.

Continuous deployment / Continuous delivery	Jatkuva julkaisu	Mahdollistaa verkkopalveluiden jatkuvan päivittämisen ilman katkoja.
Kanban board	Kanban taulu	Taulu, joka toimii tuotannon ajoitusjärjestelmänä.
JSON	JSON	Yksinkertainen tiedostomuoto tiedonvälitykseen.
Serialization	Serialisointi	Tietorakenne modifioidaan selaimelle sopivaksi.
IDE (Integrated development environment)	Ohjelmointiympäristö	Joukko ohjelmia tai ohjelma, jolla toteutetaan ohjelmistoa.

Sisällys

Liitteet

1 Johdanto

Opiskelun lähentyessä loppuaan etsin mahdollista viimeistä haastetta, jolla kehittyä ja kasvaa ohjelmoijana. Luonnollisesti insinöörille on tärkeää oppia kirjoittamaan sujuvia raportteja ja dokumentoimaan löydöksiään tai havaintojaan, mutta tämän opinnäytetyön ydin on kuitenkin henkilökohtainen kehitys. Toiveena on, että opinnäytetyöstä olisi jatkossa hyötyä myös muille, jotka haluavat kehittää itseään ja viedä ohjelmoinnin seuraavalle tasolle.

Tässä opinnäytetyössä on tarkoituksena luoda ohjelma, joka toimii muistion kaltaisesti. Sitä on helppo käyttää, se laajenee tarvittaessa ja tietoa voidaan lisätä ja poistaa tehokkaasti ja helposti. Ohjelmalle löytyy melkein vastaavia kilpailijoita, joista suurin osa on valitettavasti maksullisia, tai liian monimutkaisia. Yleensä kilpailevia ohjelmia sitoo myös tarkat säännöt tai ne ovat sidoksissa esimerkiksi johonkin peliin. Tässä ohjelmassa pyrkimykseni on poistaa näitä sääntöjä ja antaa käyttäjälle mielivaltaisesti mahdollisuuksia luoda oma tarinansa, johon palata yhä uudestaan. Ohjelmasta olisi tarkoitus tulla web-sovellus ja mahdollisesti myöhemmin siihen liitettäisiin myös mobiilisovellus. Kyseisen ohjelman rakentamiseen olisi varmasti monia hyviä vaihtoehtoja ja ohjelmointikieliä ja näiden valitsemisestakin tulee työhön oma osionsa. Ohjelman tekeminen tulee olemaan haaste ja sen toimivuuden takaamiseksi on ohjelman rakenteiden toimittava mahdollisimman geneerisesti. Geneerinen ohjelmointi on oma haasteensa ja pakottaa ohjelmoijan ajattelemaan entistä laajemmalla skaalalla käyttäjien tekemiä päätöksiä, aihe on mielenkiintoinen ja niin laaja, että tulen omistamaan sille teoreettisen osuuden tässä opinnäytetyössä.

Ohjelmoinnissa on helppo päästä alkuun, internetistä löytyy lukemattomia määriä ohjeita ja videoita harrastusta tai uraa aloittaville. Valitettavan usein pääsee tilanteeseen, jossa seuraavan askeleen löytäminen on vaikeaa, koska materiaali loppuu kesken, siksi tässä opinnäytetyössä olen pyrkinyt keräämään useasta lähteestä ohjeita ja vinkkejä oman ohjelmoinnin kehittämiseen.

Sovellukseen antoi inspiraation kollegani Marko Vehmas, joka oli suunnitellut luovansa kyseisen ohjelman itse, mutta jakoi ideansa, kun kuuli minulta puuttuvan aiheen. Ohjelma tulee auttamaan tarinankertojia ja matkailijoita, jotka haluavat kirjata omia kokemuksiaan tai tarinoita ja palata näihin myöhemmällä ajalla.

2 Geneerinen ohjelmointi

Opinnäytetyön onnistumisen kannalta on tärkeää, että työn tuloksena valmistuvasta sovelluksesta tulisi mahdollisimman geneerinen eli yleisluontoinen. Ohjelmassa käyttäjän olisi itse tarkoitus asettaa omat rajat ja käyttötarkoitus. Kun tavoitteena on, että yksi käyttäjä voi sovellusta hyödyntäen tallentaa vaikka oman mielikuvitusvaltionsa ja sen lukuisat vuoret ja niiden eläinkunnan, kun taas toinen käyttäjä haluaa tallentaa suomen parhaimmat kalastuskohteet ja niistä saamansa saaliit, on helppo ymmärtää, että rungon muodostaminen on kriittistä onnistumisen kannalta.

Geneerisellä ohjelmoinnilla tarkoitetaan ohjelmointia, jossa funktioita ja luokkia voidaan käyttää mahdollisimman monessa tapauksessa. Ohjelmointityyli, jota kutsutaan geneeriseksi ohjelmoinniksi, syntyi ML-ohjelmointikielen yhteydessä vuonna 1973 (Stepanov & Stroustrup, 1989). Geneerisen funktion on tarkasteltava vastaanotettu parametri ja käsitellä parametria monesta eri näkökulmasta, jotta saadaan varmuus kohteen tyypistä. On monia mielipiteitä siitä, mikä ohjelmointikieli sopii kyseiseen tarkoitukseen parhaiten, lopullinen päätös kannattaa aina perustaa käyttötarkoituksen mukaan. Geneerisessä ohjelmoinnissa suunnitelman merkitys korostuu ja on huomattavasti vaikeampaa hahmottaa kaikki mahdolliset polut tai reitit, jota käyttäjä saattaa valita navigoidessaan sovellusta eteenpäin. Tässä ohjelmassa voi sovelluksen lopullinen koko paisua hyvinkin suureksi, riippuen siitä kuinka paljon käyttäjä haluaa siihen tallentaa, tästä johtuen tietokannan on pysyttävä perässä ja muovauduttava tarpeen vaatiessa.

2.1 Geneerisen ohjelmoinnin määritelmä

Termiä geneerinen ohjelmointi kuulee käytettävän useassa eri kontekstissa ja tilanteessa. Yhdistävä tekijä näissä tilanteissa on muovattavuus ja turvallisuus. Jotta aihetta saadaan

vähän supistettua niin tässä työssä geneerisellä ohjelmoinnilla tarkoitetaan datatyypistä riippumatonta geneeristä ohjelmointia. Geneerisen ohjelmoinnin yhteydessä esiintyy usein myös termi abstrakti ja tähän palaamme kappaleessa 2.2. Geneerinen ohjelmointi on tietojenkäsittelytieteen ala, joka pyrkii luomaan abstrakteja esiintymiä tehokkaista algoritmeista ja tietorakenteista. Geneerisessä ohjelmoinnissa jokainen käyttäjältä saatu data analysoidaan ja jäsennetään tarkasti. Jäsennelty tieto viedään ohjelmassa eteenpäin ja tiedon tyypistä riippuen päätetään ohjelman seuraavat vaiheet. Jäsentämisen onnistuminen on kriittinen vaihe ohjelman toimivuuden kannalta ja nostaakin kehittäjän kannalta vaikeustasoa huomattavasti. Mitä geneerisemmäksi funktiot tai luokat saadaan, sitä helpompi on sen uudelleen käyttäminen ja onnistuneella lopputuloksella voidaan funktiota käyttää uudelleen ohjelmassa.

Geneerisyydessä voidaan pyrkiä luomaan myös ohjelma, joka on mahdollista kirjoittaa uudelleen useammalla eri ohjelmointikielellä, jolloin projektin vaatiessa uutta ympäristöä, voidaan olemassa oleva koodi siirtää sinne ilman suuria muutoksia. Tämän kaltaisen geneerisyyden saavuttaminen vaatii ohjelmoijalta tietämystä ruohonjuuritason ohjelmoinnista. Monissa ohjelmointikielissä on kielelle tyypillisiä ominaisuuksia tai metodeja, jotka eivät ole yhteensopivia toisen kielen kanssa, näitä on pyrittävä välttämään ja monet ”oikotiet”, jota esimerkiksi Python-kieli tarjoaa, joudutaan jättämään käyttämättä.

2.1.1 Arvoihin ja muotoihin sidottu geneerisyys

Termi ”kovakoodaus” on monelle ohjelmoijalle tuttu. Termillä viitataan yleensä käsin kirjoitettuihin ehtoihin, jotka toteuttavat aina samat ehdot, jokaisella ohjelman suorituskerralla (Kuva 1.). Ohjelmointia aloittelevat käyttävät kyseistä ohjelmointityyliä, joka taidon harjaantuessa muuttuu geneerisemmäksi yleensä tiedostamatta.

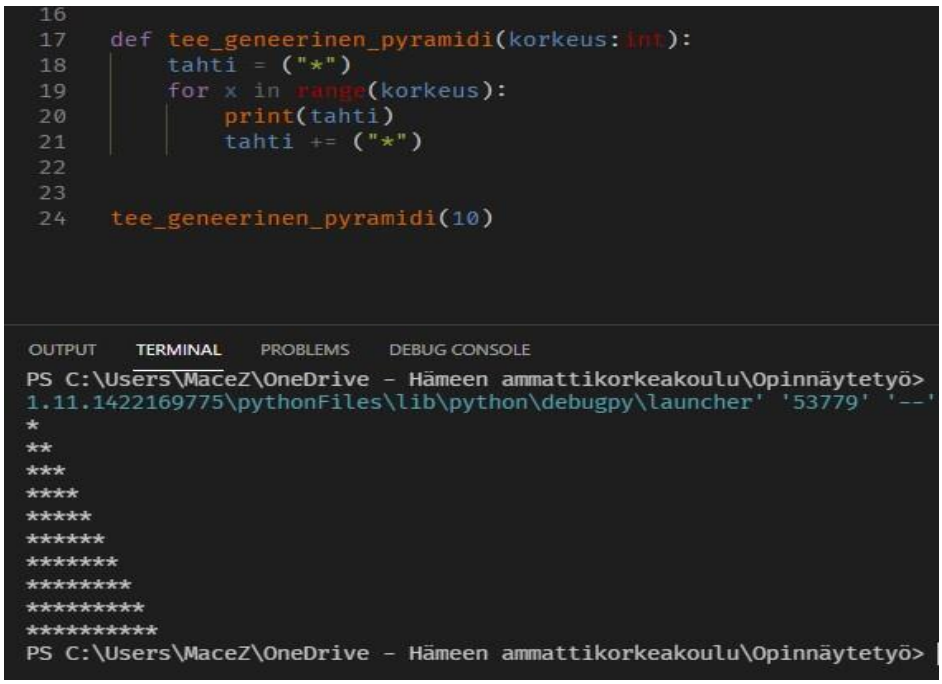
Kuva 1. Esimerkki ”kovakoodatusta” funktiosta, aina sama paluuarvo

```
2
3   def tee_pyramidi():
4       print("*")
5       print("**")
6       print("***")
7       print("****")
8       print("*****")
9       print("******")
10
11
12
13  tee_pyramidi()
14
```

```
OUTPUT   TERMINAL   PROBLEMS   DEBUG CONSOLE
PS C:\Users\MaceZ\OneDrive - Hämeen ammattikorkeakoulu\Opinnäytetyö>
1.11.1422169775\pythonFiles\lib\python\debugpy\launcher' '52858' '--'
*
**
***
****
*****
******
PS C:\Users\MaceZ\OneDrive - Hämeen ammattikorkeakoulu\Opinnäytetyö>
```

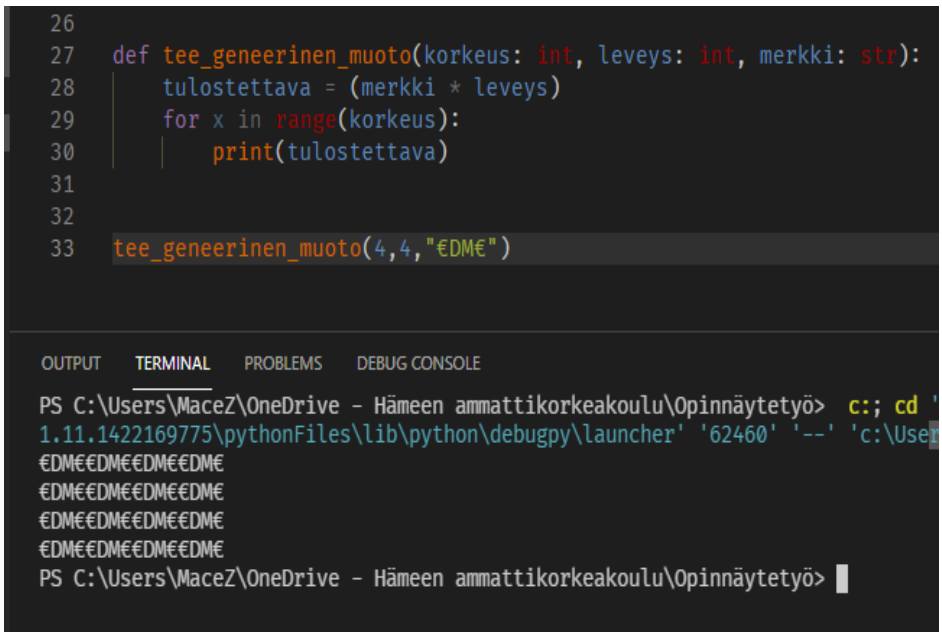
Kyseisen toiston voi korvata geneerisellä arvolla, jolloin tieto haetaan ulkoisista lähteistä tai ne annetaan käyttäjän ajaessa sovellusta, jolloin lopputulos muuttuu riippuen funktion alkuperäisistä arvoista ja saadaan muodostettua selkeä kaava lopputulokselle. Edellisen kuvan esimerkin voisi korvata geneerisellä funktiolla, joka palauttaa parametrin ”korkeus” perustuvan pyramidin (Kuva 2.).

Kuva 2. Pyramidi on geneerisempi ja hyväksyy parametrikseen korkeuden



Esimerkkiä voidaan vielä kehittää eteenpäin ja laajentaa sen käyttö hyväksymään parametreikseen korkeuden lisäksi esimerkiksi leveyden tai vaihtoehtoisen merkin ”*” sijaan (Kuva 3.). Kaikissa esimerkeissä tosin oletetaan, että funktio saa oikeat parametrit, eikä niitä jäsennetä ja sitä myöten reagoida ollenkaan.

Kuva 3. Funktion geneerisempi versio helpottaa uudelleenkäyttöä



2.1.2 Tyyppiin sidottu geneerisyys

Toinen tapa antaa ohjelmalle lisää geneerisyyttä ja muovautumiskykyä on antaa funktion käsitellä saapuneen parametrin tyyppi ja ohjata data oikealle polulle tästä riippuen. Edellisissä kuvissa nähty Python-ohjelmointikieli saattaa olla opinnäytetyön kannalta hyvä, valitettavasti se on esimerkkien kannalta huono, sillä se on dynaamisesti tyypitetty ja sidottu kieli ja parametrin tyyppi selvitetään vasta sovelluksen ollessa ajossa (Python Software Foundation, 2021). Edellisissä kuvissa nähdyt tyypit eivät ole välttämättömiä Pythonissa ja ne on lisätty raporttiin helpottamaan parametrien datatyypin ymmärtämistä, tämä ei kuitenkaan tarkoita sitä, että kyseinen ohjelmointikieli sallisi esimerkiksi kirjaimen ja numeron yhteenlaskua (Skip, 2012).

Funktion saadessa parametri, se jäsentää ja käsittelee parametrin ja toteuttaa funktion ydintoiminnallisuuden muokaten käsittelyä parametrin tyypin mukaan. Hyvänä esimerkkinä tästä on Python-kieleen sisäänrakennettu sorted-metodi. Metodi lajittelee listassa olevan datan, jonkun tietyn ominaisuuden mukaan. Esimerkiksi numerot suuruusjärjestykseen tai kirjaimet aakkosjärjestykseen.

Ollakseen geneerinen, asianomaisen metodin on osattava hahmottaa parametrin tyyppi määritteekseen, mitä erinäisiä lajitteluperusteita voidaan käyttää. Ajo on ohjattava tyyppiä vastaavaan suuntaan ja samaan aikaan funktion on pysyttävä toiminnallisuudeltaan identtisenä ja riippumattomana siitä, minkä tyyppistä dataa sille syötetään. Tätä tyypistä riippumatonta ominaisuutta kutsutaan myös usein nimellä parametrinen polymorfismi. (Cardelli, 1985)

2.2 Abstraktio ohjelmoinnissa

Abstraktion käsite nousee usein esille geneerisyyden yhteydessä. Abstraktion idean kuvaamiseen sopii arkipäiväinen esimerkki kahvinkeitosta: ”Käyttääksesi keitintä annat keittimelle sen tarvitseman veden ja pavut, lopulta käynnistät laitteen. Sinun ei tarvitse tietää kuinka laite toimii sisäisesti, tai veden lämpötilaa sen muuttuessa kahviksi”. (Thorben, 2017)

Abstraktiossa käyttäjälle annetaan yksinkertaistettu näkymä laitteen käyttöön, käyttäjän ei tarvitse tietää siitä mitä konepellin alla tapahtuu. Piilotettu logiikka voi olla hyvinkin suuri rakenne, jossa konsepti on vaikeasti ymmärrettävää, mutta käyttäjän ei tarvitse tietää siitä käyttääkseen ohjelmaa. Käyttäjälle näytetään vain se mitä vaaditaan ohjelman toimiseksi.

Abstraktio nousee usein esille geneerisyyden kanssa konseptien sisältäen joitain yhtenäisyyksiä, kuitenkin pyrkien erilaiseen lopputulokseen. Abstraktiossa tietty ryhmä voi jakaa samoja funktioita ja abstraktion tarkoitus on sitoa useampi ryhmä noudattamaan tiettyjä kaavoja. Geneerisessä ohjelmoinnissa käyttötarkoitukset voivat olla hyvin samankaltaisia tai jopa identtisiä, mutta tyypit määräävät toteutuksen nyanssit.

Tulevassa projektissa abstraktion merkitys on suuri. Ohjelman käyttäjän on saatava luoda oma maailmansa asettamillaan ehdoilla ja käyttäjältä vaaditaankin kommunikointia ohjelman kanssa. Tämän kommunikoinnin tulee olla helppoa ja virtaviivaista ja mahdollisimman pienellä määrällä rajoituksia. Ohjelmoijan kannalta kyseinen ratkaisu on kuitenkin vaikea toteuttaa. Jokainen käyttäjän antama yksinkertainen komento on ohjelmalle aluksi tuntematon. Komennon selvittämiseksi tieto joudutaan jäsentämään ja analysoimaan, jotta geneerisisiin osuuksiin ja siitä seuraavaan abstraktioon päästään. Koodin määrän pienentämiseksi, joudutaan abstraktiota hajaannuttamaan useammalle uudestaan käytettävälle tasolle, jotta ohjelma pysyy helposti hallittavissa ja ymmärrettävänä.

2.3 Miksi pyrkiä geneerisyyteen?

Aiemmissa kappaleissa havaitut esimerkit osoittavat, että geneerisyyden saavuttaminen voi paikoitellen olla hyvin hankalaa ja vaatia suurien kokonaisuuksien uudelleensuunnittelua. Ohjelmoija joutuu ajattelemaan ongelmaa useammasta eri perspektiivistä ja usein ohjelmoijien kesken nouseva ohjeen ”älä luota käyttäjiin” merkitys kasvaa. Monesti funktion muovaaminen geneerisemmäksi vaatii huomattavan määrän lisätyötä, se kuitenkin usein maksaa itsensä takaisin projektin myöhemmissä vaiheessa tai ylläpidossa.

Ensimmäiset hyödyt geneerisessä koodissa ovat heti ohjelmoijan nähtävillä vahvempien tyyppikäsittelyiden ansiota, jolloin ohjelmoija pysyy paremmin kartalla muuttujien tyypeistä

ja tämä nousee esille myös ajonaikaisissa virheissä. Koodin geneerisyyden kasvattaminen pyrkii eliminoimaan vaikeasti luettavaa ja ymmärrettävää koodia, joutumatta tekemään kompromisseja turvallisuuteen liittyen. Geneerisyyteen kannattaa pyrkiä parhaansa mukaan, jo pelkästään koodin määrän pienentymisen kannalta, joka on suora johdannainen uudelleenkäytön ansiosta. Ohjelmoijan on hyvä miettiä tulevan projektinsa vaikutusta ja sen uudelleenkäytettävyyttä, sitä kuinka paljon kyseistä koodia voisi käyttää tulevissa projekteissa, tai onko nyt suunniteltu funktio mahdollista käyttää uudelleen muodostamalla geneerisempi ratkaisu.

Suurissa projekteissa, joissa ohjelman on tarkoitus kestää pitkälle tulevaisuuteen, hyödyn määrä on entistä suurempi. Projektia ylläpitävien tahojen on helppo lisätä ohjelmaan uuttaa toiminnallisuutta, jos pohjalle on rakennettu geneerisiä kokonaisuuksia, jotka mahdollistavat uusien ominaisuuksien lisäämisen käyttäen vanhoja funktioita tai metodeja, tai estävät uusien virheiden syntymistä havaitessaan virheellisiä parametrejä vanhoissa funktioissa.

3 Tietoperusta

Teoria opinnäytetyön takana on aiheena laaja ja konseptien ymmärtäminen syvällisellä tasolla vaatisi vuosia aiheiden opiskelua, siksi konsepteihin tutustutaan tässä työssä hyvin pinnallisesti. Monia ohjelmoinnissa tärkeitä konsepteja tullaan sivuamaan ja asiasta kiinnostuneille annetaan lähteitä, joiden avulla syventää omaa osaamistaan. Tätä työtä kirjoittaessa konseptit ovat ajatustasolla tuttuja, mutta tarkoituksena on syventää näiden aiheiden osaamista ja implementoida ne käytännössä. Opinnäytetyössä luotavan ohjelman käyttötarkoitus on yksinkertainen, mutta sen saavuttaminen tulee vaatimaan suurta määrää opiskelua ja asiaan tutustumista. Valmistuvan työn onnistuminen riippuu pitkälti siitä, kuinka kykenevä se on muovautumaan vastaamaan käyttäjän tarkoitusta.

Koulusta tutuksi tulleet konkreettiset ohjelmat, joissa abstraktiota ja geneerisyyttä on luotu ohjelmallisesti, ovat antaneet ainakin teoreettisella tasolla kuvan kyseinen ohjelman olevan mahdollinen, vaikka oma taito asetetaan varmasti koetukselle. Työelämässä on valitettavan usein noussut esille ohjelmassa tehdyt liian spesifiset ratkaisut, jolloin koodin uudestaan käyttäminen ja ylläpitäminen on ollut haastavaa, jopa mahdotonta. Tällaisissa tilanteissa

uusien ominaisuuksien luominen ohjelmaan on vaatinut useita tuntia töitä ja pahimmissa tilanteissa vanhat ongelmat vain laastaroidaan, koska koodia ei uskalleta enää muuttaa, peläten sen mahdollisia vaikutuksia ohjelman muissa osissa. Työssä valmistuva ohjelma, joka kehittää ja luo itseään käyttäjien valintaan perustuen, on hyvin poikkeuksellinen ja en ole luonut aikaisemmin mitään vastaavaa.

Sovellus rikkoo paljon ohjelmistokehityksessä käytettyjä normeja, jossa ohjelmoija asettaa sovellukselleen selkeän tavoitteen ja käyttötarkoituksen, joten tiedon etsiminen tulee olemaan haastavaa ja projektia edistävät askeleet ovat pieniä. Kokonaisuuden onnistuminen riippuu pitkälti valituista ohjelmointikielistä ja kehyksistä. Tietokannan tulee olla hyvin muokattavissa ja tukea epälineaarista tietorakennetta. Käyttäjältä piilossa olevan palvelimenpuoleisen koodin on ymmärrettävä käyttäjän antamat komennot, jäsennettävä ne ja lisättävä tietokantaan mahdollisesti olemassa olevien tietojen seuraksi tai pyydettävä tietokantaa laajentumaan vastaamaan käyttäjän tarpeita. Sovelluksen toimivuus on mielestäni näiden kahden alueen vastuulla. Koska käyttäjälle halutaan vapaus luoda toivomansa kokonaisuus vailla rajoituksia, on palvelinpuolen logiikan ja tietokannan pystyttävä tulkitsemaan käyttäjän antamat toiveet ilman virheitä ja reagoitava aina täsmällisesti.

Selainpuolella tehtävä työn toteuttaminen on luultavasti helpompaa. Sovelluksen ei ole tarkoitus olla visuaalisesti näyttävä, sen on osattava vain näyttää käyttäjälle jo tiedossa oleva data ja annettava käyttäjälle mahdollisuudet lisätä uutta tietoa tietokantaan. Selainpuolella on kuitenkin tärkeää datan paikkansa pitävyys ja tämä vaatii selainpuolen, palvelinpuolen ja tietokannan jatkuvaa synkronointia, jotta tieto pysyy jatkuvasti relevanttina ja ajantasaisena. Muissa osa-alueissa joudutaan varmasti miettimään vaihtoehtoja, mutta selainpuolesta tulee todennäköisesti vastaamaan JavaScript-ohjelmointikieleen perustuva kirjasto React. Kyseinen kirjasto on hyvin suosittu ohjelmoijien keskuudessa ja sen tuore ominaisuus ”Koukut”, sekä Redux-tilanhallinta pitävät sovelluksen tilan ajantasaisena. Kirjastosta lisää osiossa Tekniikat.

4 Kehittämistyön päämäärä ja tarkoitus

Kehitettävän sovelluksen ja tämän opinnäytetyön tarkoitus on ensisijaisesti kehittää itseäni ohjelmoijana ja ymmärtää usean tutuksi tuleen konseptin mahdollisia käyttötarkoituksia ja teorioita niiden taustalla. Ohjelmistokehityksessä on helppo ottaa ensimmäiset askeleet materiaalin suuren määrän ansiosta, mutta laajemman ymmärryksen saaminen jää usein vajavaiseksi materiaalin puuttumisen johdosta. Teknologia-ala kehittyy todella nopeasti ja uusia innovaatioita syntyy päivittäin, silti tietyt konseptit säilyvät ja näiden konseptien ymmärtäminen syvemmällä tasolla auttaa ohjelmoijaa näkemään mahdollisiin ongelmiin useampia vaihtoehtoja. Valitettavan useat konsepteja syvemmin tutkivat kirjat ovat maksullisia ja luotettavien lähteiden löytäminen mielipiteiden joukosta käy vaikeammaksi. Pyrin dokumentoimaan sovellusta tehtäessä eteen tulleet ongelmat mahdollisimman selkeästi, jotta kyseisestä aiheesta kiinnostuneet voivat myös syventää osaamistaan ja mahdollisesti löytää tekstistä ratkaisun omien ongelmiensa selvittämiseen.

Työn toimeksiantaja kertoi, että vaikka kilpailevia ohjelmia on olemassa, ne ovat liian sidottuja johonkin tiettyyn sisältöön, eivätkä anna käyttäjälle tarpeeksi vapauksia. Useat kilpailevat ohjelmat ovat korkean kuukausimaksun takana, joten työstä on mahdollista jopa saada pientä passiivista tuloa, riippuen lopputuloksen onnistumisesta ja tämä tietysti motivoi opiskelijaa omalta osaltaan. Toimeksiantaja ilmoitti myös kiinnostuksensa ylläpitämään ohjelmaa mahdollisuuksien mukaan.

Tavoitteena on luoda helposti ylläpidettävä sovellus joka sääntöjen puuttumisen ansiosta olisi käytettävissä suuremmalle käyttäjäkunnalle kuin on odotettu. Ideaalinen tilanne olisi sovellus, jota kaikki voisivat käyttää vaihtelevissa määrin. Sovelluksen olisi oltava helposti ylläpidettävä ja sovelluksen rungon olisi mahdollistettava sen autonominen laajeneminen riippuen käyttäjän vaatimuksista.

5 Suunnittelu

Hyvä suunnitelma on ohjelmoijalle kuin huijauskoodi videopeliin. Se auttaa pääsemään vaikeiden kohtien yli ja jopa mahdollistaa maaliin pääsemisen. Projekteissa kuten tässä

opinnäytetyössä, jossa tuleva projekti on vain teoreettisella tasolla, eikä ole käytännön kokemusta vastaavasta työstä, sen merkitys korostuu entisestään. Sovelluksen todennäköisesti toteuttaa ohjelmoijan lempiohjelmointikielellä ja tutuilla työtavoilla, mutta tämä ei välttämättä johda siihen, että lopullinen tuotos olisi hyvä tai edes toimiva. Aikaa voi kulua turhaan kohtiin, jotka eivät vain ole mahdollista toteuttaa ohjelmoijan tutuiksi tulleilla tekniikoilla ja turhautuneisuus kasvaa. Lopulta deadlinen potkiessa päähän, tehdään hätiköityjä päätöksiä ja saadaan epäonnistunut lopputulos.

Suunnitellessa laajempaa kokonaisuutta on hyvä luoda edes minimaalinen arkkitehtuuri tulevasta sovelluksesta. Ohjelman jokaisesta osa-alueesta kartoitetaan sen vaatimukset ja tarpeet, sekä luetellaan mahdolliset vaihtoehdot toteuttaa kyseinen osa-alue. Jokainen ohjelmoija voi kertoa oman luotto-ohjelmointikielen ja miksi se sopii kaikkiin käyttötarkoituksiin, mutta jos projektilla ei ole kiire, niin olisi ehdotonta tutustua muihin kieliin ja listata jokaisen vahvuudet ja heikkoudet tulevan projektin näkökulmasta.

Tätä opinnäytetyötä suunnitellessani listasin projektin tärkeimmät kohdat, ja niiden vaatimukset:

- Tietorakenne. Datan oltava helposti selattavissa ja laajennettavissa.
- Tietokanta. Tukee tietorakennetta ja pysyy muutosten perässä.
- Palvelinpuoli. Tukee abstraktiota ja geneeristä ohjelmointia.
- Selainpuoli. Käytettävyys ja pysyy tietokannan kanssa synkronoituna.

Nämä neljä aluetta tulevat toimimaan projektin selkärankana ja jokaiseen kohtaan valitaan siihen mielestäni parhaiten sopiva ohjelmointikielen ja kehyksen yhdistelmä. Mahdollisia tähän projektiin sopivia kirjastoja ja kokoelmia etsitään ja hyödynnetään.

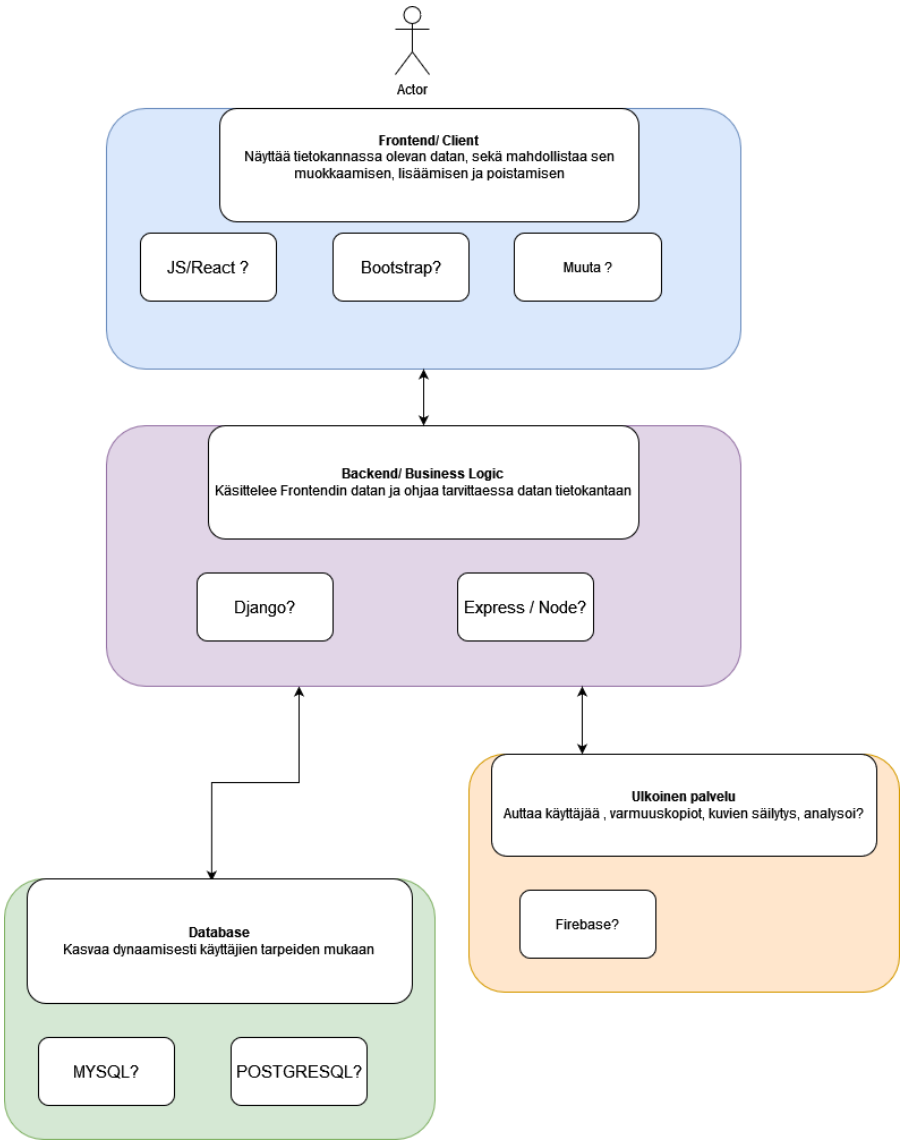
Suunnitellessa projektia, jonka tuleva sisältö on ohjelmoijalle itselleen tuntematonta aluetta, on hyvä varata reilusti enemmän aikaa jokaisen kohdan suunnitteluun. Tunnit kuluvat nopeasti uusia aiheita tutkiessa ja usein edellisen päivän aikana saadut tulokset ja oppi osoittautuvat seuraavana päivänä löydetyn ratkaisun edessä huonoksi vaihtoehdoksi. Nämä takaiskut kannattaa pitää positiivisena ja muistaa että oma tietämys kasvaa jokaisen luetun

artikkelin tai katsotun opetusvideon jälkeen. Sovellus, joka kasvattaa itseään tuntuu minun kokemukseni varjolla mahdottomalta idealta, ja tiedän että nykyinen osaamiseni ei riitä sovelluksen toteuttamiseen, mutta eivätkö kaikki teknologiaa mullistaneet ideat ole aluksi kuulostaneet mahdottomilta?

5.1 Arkkitehtuuri

Pyöriteltyäni arkkitehtuuria mielessäni useamman päivän, päätin luoda alustavan ja hyvin karkean luonnoksen arkkitehtuurista nykyisillä tietotaidoillani. Arkkitehtuurin olisi tarkoitus laajentua ja tarkentua oman tietämykseni mukana ja lopulta se saisi lopullisen muotonsa, projektin lähentyessä loppuaan. Luonnoksen tekemiseksi päätin käyttää jo ennestään tuttua Draw.io ohjelmistoa. Tarkoituksena on luoda kokonaisvaltainen arkkitehtuuri, jossa näkyy ohjelma pääpiirteittäin (Kuva 4.). Kokonaisvaltaisen arkkitehtuurin lisäksi yritin luonnostella myös tarkemman SQL-kaavan, jossa näkyisi tietokanta tarkemmin, mutta koska sääntöjä ei pitänyt luoda, päätin antaa asian vielä olla. Ohjelman idea on kuitenkin siinä, että käyttäjä luo omat tietokantansa vastaamaan tarpeitaan.

Kuva 4. Alustava arkkitehtuuri sovelluksesta.



5.2 Tietorakenne

Tietojenkäsittelytiede pyörii pitkälti tiedon ja datan analysoiminen, näyttämisen ja tulkitsemisen ympärillä. Moniin kysymyksiin ei riitä pelkkä tieto vastauksesta, vaan vastauksen saamiseksi tieto on järjestettävä oikein ja saatava esille nopeasti, jotta muodostuu kokonaisvaltainen vastaus kysymykseen. Monen modernin ohjelman tarkoitus on saada tämänkaltaisiin kysymyksiin vastaus ja vastauksen pitää tulla niin nopeasti, kuin mahdollista. Tässä opinnäytetyössä valmistuva ohjelma on hyvä esimerkki ohjelmasta, jossa vastauksen etsiminen ja saaminen on tultava nopeasti ja tarkasti. Ohjelmaa käyttävä henkilö

saattaa luoda hyvinkin monimutkaisen kokonaisuuden, jossa dataa säilötään usealla eri tasolla ja näiden tasojen välillä hyppiminen voisi turhauttaa käyttäjää, jos tietojen saamisessa kestäisi kauan.

Tietokoneiden kehittyessä yhä tehokkaammaksi ja prosessointitehojen kasvaessa voisi kuvitella tämän datan noutamiseen käytetyn ajan olevan niin pieni, että käyttäjä ei edes huomaisi odottavansa. Monesti pienissä ohjelmissa datan organisoiminen onkin irrelevanttia siitä yksinkertaisesta syystä, että dataa on vähän. Ohjelmoijan olisi kuitenkin hyvä nähdä asiat pitemmällä tähtäimellä. Voiko kehitettävässä ohjelmassa joskus olla kehittyneempi visuaalinen ilme tai vaikeampia laskutoimituksia, jotka mahdollisesti syövät tietokoneelta enemmän tehoja? Teknologian ottaessa suuria harppauksia, myös tietokoneilta vaaditut asiat muuttuvat monimutkaisemmiksi, siksi tiedon järjestäminen suunnitellusti tulee aina olemaan tärkeä aspekti tietokoneohjelmia.

Tietorakenteissa pitää myös huomioida, että ei ole yhtä tapaa järjestää data, joka olisi toista parempi. Jokaiselle kehitetylle rakenteelle löytyy selkeä tarve, joka on johtanut kyseisen rakenteen muodostumiseen. Siksi jokaisen ohjelman kohdalla, jonka ohjelmoija luo, on syytä miettiä paras tapa järjestää tieto, jotta sen saaminen on mahdollisimman tehokasta. Jokaisella askeleella, joka rakenteessa otetaan, on hintansa. Tiedon noudon kustannusten hallitseminen antaa ohjelmoijalle hyvän pohjan tuleviin ohjelmiin.

Tietorakenteen ymmärtämisen helpottamiseksi kuvitellaan jokainen kirjoitus tai artikkeli, jonka käyttäjä kirjoittaa, yhtenä datapisteenä. Saadessani projekti-idean työkaveriltani hänen visiossaan oli selkeästi lähtökohta, josta ohjelman oli tarkoitus alkaa. Käyttäjän olisi luotava aloituspiste, josta data haarautuisi käyttäjän haluamalla tavalla, jokaisella datapisteellä olisi oma paikka rakenteessa ja yksittäisten pisteiden luominen olisi sovelluksen konseptia vastaan. Jokaisella datapisteellä olisi siis oltava ainakin yksi vanhempi. Datapiste voi olla datan päätepiste tai sillä voi olla yksi tai rajaton määrä lapsia. Työn tilanneen kollegani kuvaus vastaa pitkälti puutyylistä tietorakennetta. Datarakenteita on monia erilaisia, ne voidaan esimerkiksi lineaarisiin ja epälineaarisiin rakenteisiin.

5.2.1 Lineaariset tietorakenteet

Lineaariset tietorakenteet tulevat ohjelmointia aloittaville nopeasti tutuiksi esimerkiksi listojen muodossa, vaikka ohjelmoija itse ei olisikaan tietoteinen kyseisestä termistä. Lineaarinen tietorakenne on tietorakenne, jossa järjestetään listaan tai peräkkäiseen järjestykseen, kuten yksiulotteiseen taulukkoon (Puntambekar, 2020, s. 33).

Tunnettuja lineaarisia tietorakenteita ovat esimerkiksi:

- Listat
- Pinot
- Linkitetyt listat
- Jonot

Jokaisessa rakenteessa tieto on järjestetty lineaarisesti, eli jokainen datapiste on liitoksissa edelliseen ja seuraavaan datapisteeseen. Data on suorassa linjassa, eli tietopisteiden välillä on vain yksi polku (Kuva 5.). Lineaariset tietorakenteet ovat yksinkertaisia ja helppoja implementoida, tästä syystä niitä löytyykin melkein jokaisesta tietokoneohjelmasta.

Kuva 5. Lineaarisia tietorakenteita

5.2.2 Epälineaariset tietorakenteet

Opinnäytetyöni ohjaajalta Petri Kuittiselta sain ohjeet materiaaliin, joka hänen mielestään oli tutustuminen arvoinen työhöni liittyen. Kyseessä oli Graafi-niminen tietorakenne. Kyseinen rakenne kuuluu epälineaarisiin tietorakenteisiin ja on jo huomattavasti lineaarisia tietorakenteita vaikeampi kokonaisuus toteuttaa. Usein lineaarinen rakenne ei yksinkertaisesti riitä tuottamaan toivottua lopputulosta ja silloin on käytettävä epälineaarista vaihtoehtoa.

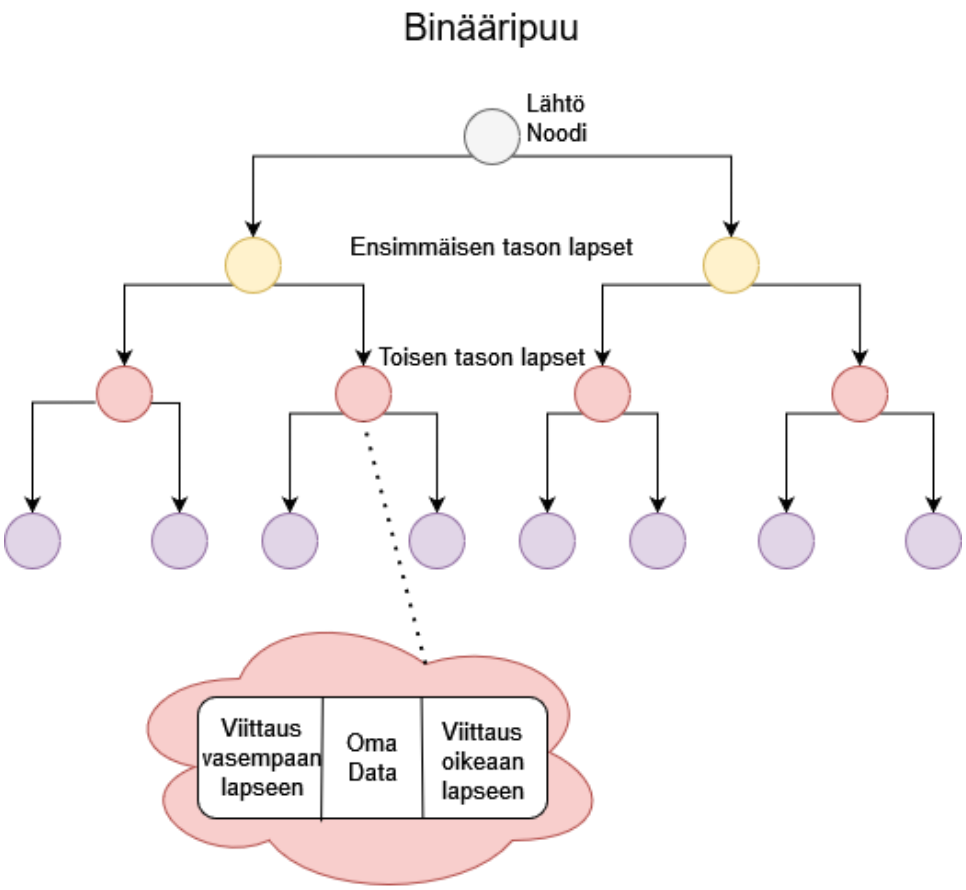
Lineaariset rakenteet muodostavat putken tai suoran näköisen rakenteen, kun taas epälineaarisessa rakenteessa datapisteiden välillä voi olla useampia polkuja ja järjestys ei ole peräkkäinen. Monen tunnetun epälineaarisen rakenteen voi mieltää olevan Graafi-

rakenteen rajoitettu versio, jossa datapisteiden välisiä polkuja on rajoitettu noudattamaan tiettyä kaavaa. Tunnetuin näistä on binääripuu. Epälineaarisessa rakenteessa data ei ole perättäistä ja dataa ei pysty käymään läpi vain yhdellä ajolla, koska tieto voi haarautua monelta osin.

5.2.3 Binääripuu

Binääripuu on saavuttanut suuren suosion ohjelmoijien keskuudessa. Se on teoriassa yksinkertainen, mutta sen implementoiminen voi paikoitellen olla hankalaa. Binääripuu mielletään ylösalaisin käännetyksi puuksi, jossa jokaisella datapisteellä täytyy olla yksi vanhempi ja enintään kaksi lasta (Kuva 6.). Datapisteet sisältävät siis oman datansa ja viittaukset vasempaan ja oikeaan lapseen.

Kuva 6. Binääripuu, sen datapisteet ja polut.



Binääripuu voidaan jakaa useampaan pienempään alapuuhun, jossa puu jaetaan useaksi pienemmäksi puuksi. Edellisessä kuvassa näkyvä korkein datapiste tunnetaan juurena. Pisteet, joilla on sama vanhempi, kutsutaan sisaruksiksi. Yhteisen isovanhemman jakavat pisteet ovat serkkuja ja datapisteet, joista ei jatku enää polkuja tunnetaan lehtinä. Tilanteissa, joissa lapsia on useampi kuin kaksi rakennetta kutsutaan vain Puu-rakenteeksi. Binääripuuta luodessa kannattaa muistaa luomisen vaikeus, herkkä alttius virheille ja sen yhteensopimattomuus muistihierarkian kanssa (Saikkonen & Soisalon-Soininen, 2008).

Binääripuusta on monia nyansseja, joilla tarkennetaan binääripuun tilannetta, kuten täydellinen binääripuu, jossa jokaisessa datapisteessä on kaksi lehteä ja kaikki datapisteet ovat samalla korkeudella. Hyvin toteutettu binääripuu on tehokas työväline datan löytämiseen ja kyseinen rakenne on niin tunnettu ja tehokas, että monet työpaikat käyttävät sitä testaamaan uusien työntekijöiden taitotasoa.

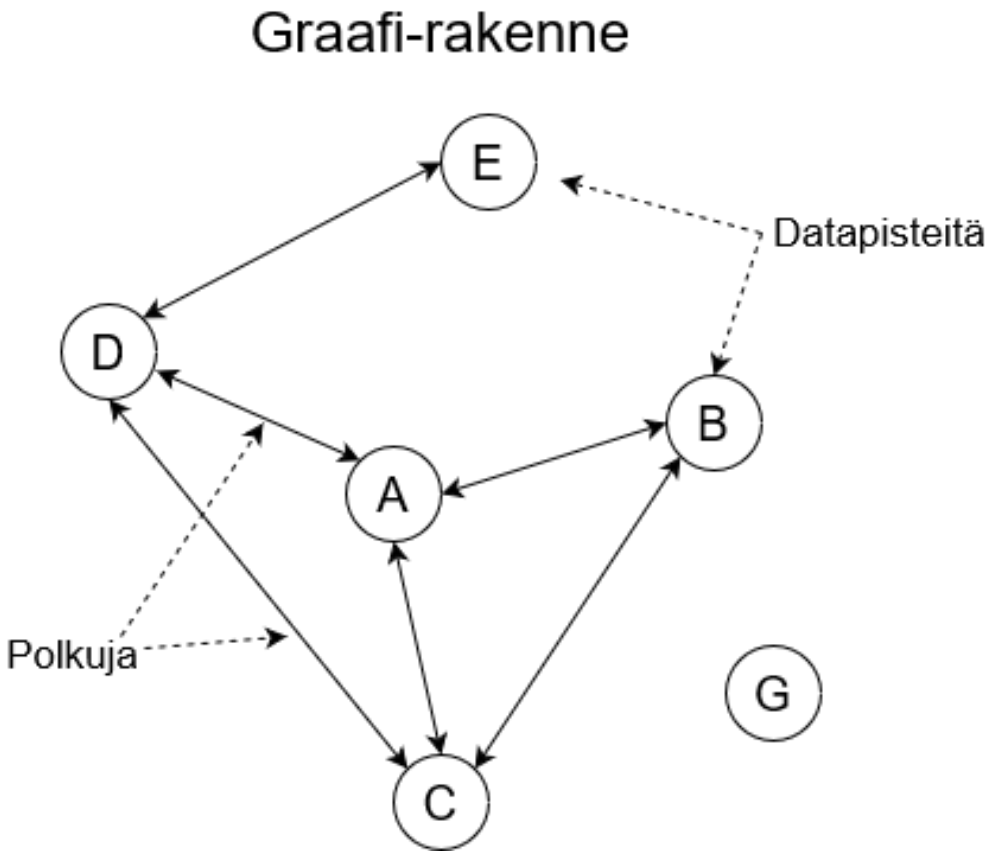
5.2.4 Graafi

Graafi-tietorakenne sisältää kokoelman datapisteitä ja ne ovat liitoksissa toisiin pisteisiin. Rakenteen ymmärtämisen helpottamiseksi käytetään yleensä esimerkkinä sosiaalista mediaa ja niissä usein esiintyvää kaveriverkostoa, jossa jokainen ihminen edustaa datapistettä. Yleensä molemmat kaveruussuhteessa olevat pitävät toisiaan kavereina ja tähän väliin muodostuu polku, jossa tieto voi kulkea kumpaankin suuntaan. Tässä suuressa verkostossa muodostuu lukematon määrä polkuja, jotka sitovat datapisteet eli kaverit toisiinsa (Kuva 7.). Graafi on myös suuresti käytössä matematiikassa ja se tunnetaan matemaattisesti ilmaistuna järjestettynä parina.

Rakenteessa on käytössä suuri määrä termejä käytössä, joista moni tukee rakenteen ymmärtämisen helpottamista. Läheiset pisteet ovat niitä, joita yhdistää polku. Polku itsessään voi tarkoittaa matkaa useamman eri pisteen kautta tiettyyn datapisteeseen. Graafit, joissa poluilla ei ole suuntaa kutsutaan suunnattomaksi graafiksi ja suunnatuksi graafiksi kutsutaan rakennetta, jossa datalla on tietty suunta. Suunnatuissa graafeissa polku (a, b) ei tarkoita, että olisi myös polku (b, a). Graafi-rakenteessa on usein tapana mitata

pisteiden välistä etäisyyttä, niiden välissä olevien pisteiden lukumäärällä ja tämä auttaa usein ohjelmoijaa hahmottamaan rakenteen suuruuden.

Kuva 7. Graafi-rakenne.



Graafi voi olla hyvin vaikea liittää osaksi omaa ohjelmaa, sillä graafien muodostamat mallit voivat paisua todella suureksi, siksi malliin päädyttäessä omassa ohjelmassa täytyy harkita sen tuomat hyödyt tai haitat. Monet epälineaariset rakenteet ovat vain rajoitettuja versioita graafista, kuten edellä mainittu binääripuu ja ovatkin huomattavasti helpompia toteuttaa.

Tässä opinnäytetyössä toivon binääripuun riittävän kuvaamaan datapisteiden välistä relaatiota ja tämä oli myös toimeksiantajan näkemys tulevan ohjelman toiminnasta. Jos tämä ei riitä, siirryn käyttämään Graafi-rakennetta.

5.3 Tietokanta

Opinnäytetyössä tullaan tarvitsemaan luonnollisesti myös tietokantaa, johon käyttäjien kirjoitukset säilötään. Tietokannan olisi hyvä tukea mahdollisimman monimutkaisia tietorakenteita, jotta ohjelmaa luodessa voitaisiin pitäytyä alkuperäisessä suunnitelmassa. Ohjaajani Petri Kuittinen antoi tähänkin hyvän suunnan ja kertoi PostgreSQL-nimisen tietokannan tukevan graafeja ja siitä suppeampaa binääripuuta.

Erilaisia tietokantoja on jo lukematon määrä ja niiden toiminnallisuudet ovat hyvin monipuolisia. Graafi-rakenteen tukeminen tietokannoissa on viimeisen vuosikymmenen aikana kokenut eksponentiaalisen kasvun, johtuen sosiaalisesta mediasta ja sen suosion noususta.

Microsoftin kehittelemä SQL Server on vuodesta 2017 asti tukenut osittain graafeja, lisäämällä MATCH-avainsanan SQL ohjelmaansa (Microsoft, 2021). Tämän jälkeen on lisätty myös muita ehostuksia, nämä päivitykset eivät kuitenkaan anna täysimittaista tukea graafien implementoimiseen.

Neo4j graafitietokanta on suurta suosiota saavuttanut kokonaisvaltainen työkalu, joka on keskittynyt graafien tallentamiseen ja tietokantojen luomiseen käyttäen kyseistä tietorakennetta. Neo4j on natiivi graafikanta, eli se poikkeaa paljon perinteisistä relaatiokannoista. Tietokanta etsii datapisteiden polut tehokkaasti, koska jokainen tietopiste sisältää itsessään viittaukset omista läheisistään, joten polut tiedetään jo kyselyä tehtäessä, tämä johtaa nopeampiin hakutuloksiin tilanteissa, joissa polut muodostavat pitkiä ketjuja. Tämän hyödyn kääntöpuolella on kuitenkin tietokannan vaatima muisti (Weinberger, 2018).

Vuonna 1995 julkaistu PostgreSQL on vuosien aikana kehittynyt hyvin suosituksi vapaan lähdekoodin tietokantajärjestelmäksi. Vaikka kyseessä on relaatiotietokanta niin PostgreSQL tukee suoraan graafeja ja sille on suunniteltu monia tietokannan hallintatyökaluja, jotka auttavat työskentelemään graafidatan kanssa. PostgreSQL tukee myös monin tavoin GraphQL-datakyselykieltä. Datakyselykieli mahdollistaa datan tarkan noutamisen tilanteissa, joissa sovellus on paisunut isoksi ja halutaan välttää palvelimenpuolelta saapuvia ylinoutoja.

Tietokannat kuten MongoDb luokitellaan NoSQL-tietokannoiksi. Kyseiset tietokannat poikkeavat perinteisistä tietokantaratkaisuista, koska niillä ei ole tiukasti määrättyä skeemaa. NoSQL-kannat on suunniteltu tiheisiin muutoksiin tietokannoissa, esimerkiksi tilanteissa, joissa tietokantaan halutaan lisätä lisää uusia rivejä alkuperäisen luomisen jälkeen. Yleinen harhaluulo, että relaatiotietokannat tukevat paremmin toisiinsa sidoksissa olevaa dataa, on väärä, dataa vain tallennetaan eri tavoin (Schaefer, n.d). NoSQL-tietokantoja voi olla useaa eri tyyppiä kuten:

- Dokumentti -tietokantoja, jotka tallentuvat JSON objekteihin
- avain-arvo -tietokantoja, joita käytetään yksinkertaisessa rakenteessa
- laajasarake -tietokantoja, missä kolumnit luodaan dynaamisesti
- graafi -tietokantoja, joissa data säilötään datapiseinä ja polkuina

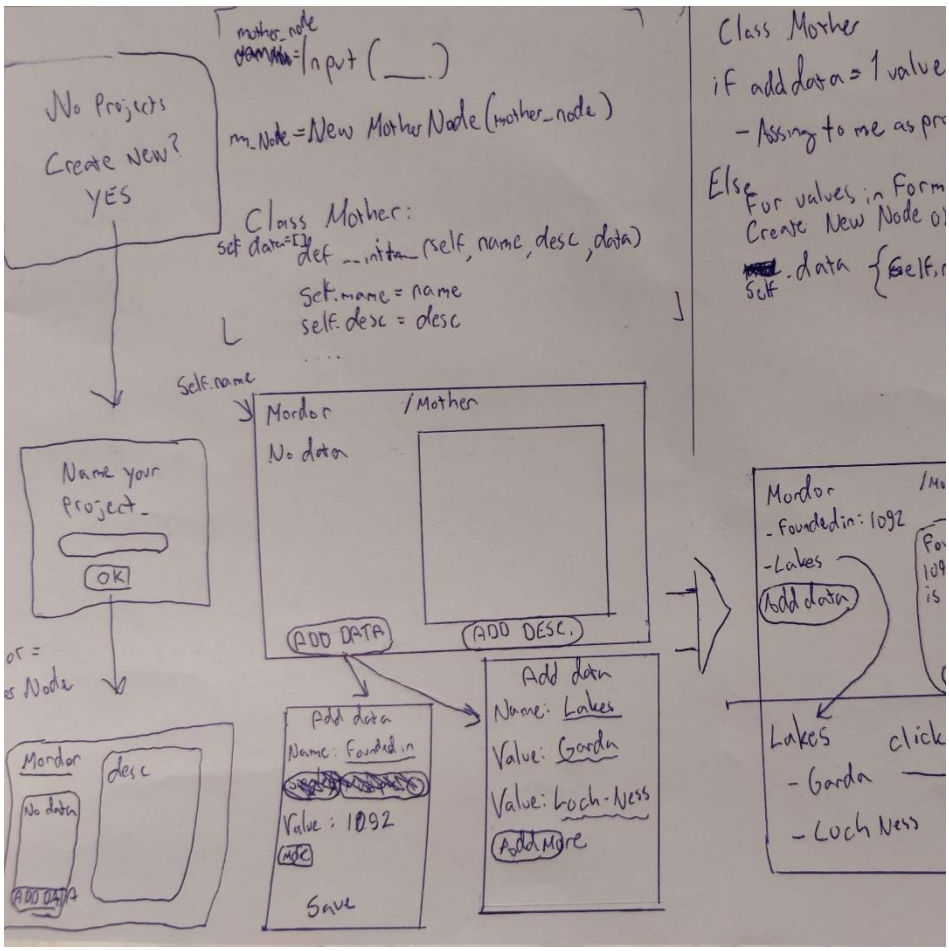
Opinnäytetyössä muodostuvan ohjelman käytön aloituksessa tullaan tietokanta pitämään tyhjänä, mutta sen pitää pystyä kasvamaan käyttäjän kanssa, tästä syystä edellä mainittu NoSQL-tietokanta päätyy todennäköiseksi valinnaksi tulevassa työssä.

5.4 Selainpuolen käyttöliittymä

Opinnäytetyössä toteutettava verkkosivulla toimiva käyttöliittymä tulee olemaan single-page application. Käyttöliittymän luontiin käytettyä aikaa ei priorisoida visuaaliseen aspektiin, vaan käyttäjälle halutaan antaa mahdollisimman intuitiivinen kokemus, jossa tiedon lisääminen on helppoa ja suoraviivaista. Kirjoitettuun tekstiin saadaan upotettuja linkkejä, joista päästään seuraavaan aiheeseen tai tarkennettuun artikkeliin. Käyttäjälle annetaan mahdollisuus palata tekstissä taaksepäin murupolulla ja tämän toteuttaminen on helppoa puurakenteen ansiosta. Jokaista kirjoitettua tekstiä täytyy päästä editoimaan nopeasti ja päivitetty tieto on välittömästi tallessa palvelimen puolella. Käyttäjälle täytyy antaa myös vaihtoehto luoda listoja suuremmista tietokokonaisuuksista, ja tämä lomake, jonka käyttäjä täyttää, tulee aukeamaan käyttöliittymästä löytyvästä napista. Edellä mainitun kokonaisuuden suunnittelu osoittautui hankalaksi toteuttaa mielikuvilla, joten käyttöliittymä

ja sen kanssa toteutuvasta vuorovaikutuksesta piirrettiin kuva auttamaan hahmottamista (Kuva 8.).

Kuva 8. Karkea piirros käyttöliittymästä ja logiikasta.



Karkeaa piirustusta luodessa oli jo selkeä visio selainpuolella toimivista tekniikoista, ja kuvissa näkyvät nuolet eivät merkitse siirtymistä toiselle sivulle, ne osoittavat uudelleen piirtämistä. React on monen modernin ohjelmistokehittäjän valinta, kun halutaan luoda dynaamisesti päivittyviä sivuja. Kielen konsepti pyörii paljon komponenttien ympärillä, joista tehdään mahdollisimman geneerisiä uudelleenkäytettävyyden vuoksi. Kyseinen kieli venyy monipuolisiin suorituksiin oikeissa käsissä ja tulee toimimaan tämän opinnäytetyön selainpuolen kirjastona.

5.5 Palvelinpuolen logiikka

Palvelinpuolelta tullaan vaatimaan mahdollisimman paljon muovautumiskykyä. Palvelinpuoli joutuu analysoimaan kaikkia käyttäjän antamia komentoja, jotka yleensä tulevat tekstinä, joten tekstinkäsittelyä helpottavien ohjelmointikielien käyttö on välttämätöntä. Palvelinpuolelta oli olennaista etsiä myös kehyksiä, joilla olisi mahdollisimman paljon kirjastoja, joita voisi hyödyntää bisneslogiikan luomiseksi, jotta aika ei kuluisi täysin sen suunnittelemiseen.

Palvelinpuolta päättäessä on huomioitava haluttu nopeus, datan luotettavuus ja sen laajenemiskyky. Opinnäytetyössä, jossa käsitellään tekstiä kirjoittajan määräämään tahtiin nopeus saa jäädä pienemmälle huomiolle. Käyttäjälle näkyvän datan täytyy olla ajantasaista, joten luotettavuuteen on kiinnitettävä huomiota ja sovelluksen kasvupotentiaalin ollessa rajaton palvelimen on pystyttävä kasvamaan tarpeen vaatiessa. Opinnäytetyöllä on myös marginaalinen mahdollisuus menestyä ohjelmistomarkkinoilla, joten turvallisuusaspektiin käytettiin enemmän aikaa.

Palvelinpuolen kehyksessä tulisi olla mahdollisimman monta sisäänrakennettua suojaa yleisimpiä hyökkäyksiä vastaan. Palvelinpuolen turvaaminen on itsessään niin suuri aihe, että alueen ymmärryksen nostaminen vaaditulle tasolle vaatii ohjelmoijalta useampaa tuntia opiskelua.

6 Tekniikat

Työhön valitut tekniikat ovat pitkän suunnittelun tulosta ja kyseisiä tekniikoita päättäessä valitsin ne, jotka koin sopivan työhöni. Tekniikoiden valitsemisessa kului myös tavallista enemmän aikaa työn geneerisestä luonteenpiirteestä johtuen. Työn edetessä huomattiin, monissa kohdissa päätöksien olleen hätäisiä ja muutoksia jouduttiin tekemään. Pyrin kuitenkin noudattamaan alkuperäistä suunnitelmaa luomatta liikaa sääntöjä ohjelman ytimelle.

6.1 Frontend

Selainpuolelta ei vaatimuksia löytynyt ja tästä syystä siihen valikoitui JavaScript-ohjelmointikielelle luotu React-kirjasto. Kyseinen kirjasto on tämän päivän markkinoilla todella kovassa suosiossa ja siksi kyseisten taitojen ylläpitäminen on tärkeää. Tähän projektiin haluttu Single-page app tyyli on myös helppo toteuttaa kirjastolla ja ohjelmasta halutaan moderni. Kirjasto on myös erittäin yhteensopiva monen muun teknologian kanssa ja antaa vapaudet valita projektin palvelinpuolelle sopivimman kehyksen.

6.1.1 React.js

Vuonna 2013 JavaScript-kielelle suunniteltu React-kirjasto on nostanut suosiotaan tasaisesti. Kirjastoa ylläpitää Meta-niminen yhtiö, sekä yksittäisiä kommuuneja ja ohjelmoijia. Kirjasto on suunniteltu yhden sivun applikaatioiden käyttöliittymää varten ja sen tavoite on vain ylläpitää sivuston näkymän tilaa (Pandit, 2021). Kirjasto antaa ohjelmoijalle mahdollisuuden luoda sovelluksia, jotka lataavat sivun sisällön muistiin ja päivittävät sivua dynaamisesti käyttäjän kanssa vuorovaikutuksessa.

Kirjasto luottaa komponentteihin ja sivusto luodaan, leikkaa ja liimaa -periaatteella, jossa ohjelmoija yrittää luoda mahdollisimman yleiskäytännöllisiä komponentteja, joita voi mahdollisesti uudelleenkäyttää sivuston laajentuessa. Sivustopohjat luodaan käyttäen JSX-syntaksia, joka on JavaScriptin syntaksin laajennus, ja käyttää HTML-syntaksista tuttuja elementtejä. Kirjastolla on myös omia natiivikirjastoja, joita hyödyntäen kirjastolla voidaan kehittää mobiilisovelluksia.

JavaScriptiä käyttäneelle ohjelmalle kirjaston yksinkertaisten komponenttien luominen on suoraviivaista, mutta oppimiskäyrä kirjaston tehokkaimpien ominaisuuksien käyttöönottoon on jyrkkä ja vie aikaa. Kirjaston pohjustaminen vie aikaa ja tämän johdosta yksinkertaisissa sivuissa kirjaston käyttö voi olla yliammuttua.

6.1.2 JavaScript

JavaScript on jokaiselle käyttöliittymää tehneelle ohjelmoijalle tuttu. Kieli on teknologisessa mittapuussa jo hyvin vanha ja se on kokenut merkittäviä muutoksia vuosien aikana. Netscapen kehittämä dynaaminen komentokieli suunniteltiin lisäämään toiminnallisuutta ja elävyyttä verkkosivuille. Kieli ei sijoitu vain selain- tai palvelinpuolelle vaan se tarvitsee isäntäympäristöä kuten selainta, edellä mainitussa tapauksessa se kuuluisi selaimen puolelle, mutta on tarvittaessa kykenevä toimimaan molemmissa. Kielen toiminnallisuus ja suosio on antanut myös loistavan työkalun hakkereille, jotka valjastavat kielen puolelleen selainten ajaessa kyseenalaista koodia päivittäin (Musch & Johns, 2021).

Kielen suurin poikkeavuus on sen prototyypitetty periminen. Jokainen kielellä luotu objekti perii joltain toiselta objektilta ja tämän voisi mieltää luokkien vastakohdaksi. Vaikka JavaScriptin selkeä painopiste on selainpuolen toiminnallisuus, löytyy kielelle monta eri käyttötarkoitusta, kuten pelien kehitys ja mobiilisovellukset. Web-sovelluskehityksen pyhän kolminaisuuden muodostavat HTML, CSS ja JavaScript. Perinteisessä verkkosivussa HTML-merkintäkieli toimii runkona, CSS-tyylisivu vastaa sivun ulkonäöstä ja JavaScript antaa sivulle liikettä ja elävyyttä.

JavaScript jakaa hyvin paljon mielipiteitä ohjelmoijien kesken. Siinä missä toiset hehkuttavat kielen yksinkertaisuutta, moni vihaa ratkaisuja, joita kieltä kehittäessä on otettu. Monesti nousee esille kielen turvallisuus: sitä on helppo väärinkäyttää, sillä se on näkyvissä käyttäjälle. Kieltä käyttäessä muodostuvia virheitä on vaikea jäljittää ja se tukee vain singulaarista perimistä. Jokaiselle heikkoudelle löytyy luonnollisesti hyviä puolia: suosionsa ja tukensa lisäksi JavaScript on erittäin yhteensopiva monen muun ohjelmointikielen kanssa.

6.2 Backend

Palvelinpuolelle oli monta hyvää vaihtoehtoa ja kyseisen alueen tekniikoiden päättäminen oli vaikein päätös sovelluksen osalta. Sovelluksessa haluttu dynaamisuus vaatii venyvää tietokantaa kuten Mongo. Datarakenne puolestaan olisi suosinut graafipohjaista kantaa, joita on luotu rakenteen tallentamiseen. Taustalla oli kuitenkin vahva halu sitoa palvelinpuoli

käyttäen Django-kehystä, joka on osoittautunut aikaisemmissa töissä toimivaksi. Tähän sovellukseen valitut palvelinpuolen tekniikat ovat modernin teknologian aallonharjalla. Monet päätyvät valitsemaan selainpuolessa toimivan React-kirjaston kanssa Express-kehykseen kääritytyn Node.js palvelimen pitääkseen sovelluksessa käytettävän kielen yhtenäisenä. Opinnäytetyöhön kuitenkin valikoitui Django-niminen verkkokehys, joka toimii hyvänä vaihtoehtona puhtaan suunnittelunsa ja skaalautuvuutensa johdosta. Tietokannan hallinnasta vastaa PostgreSQL, sillä se sopii hyvin yhteen graafi-tietorakenteen kanssa ja tarjoaa miellyttävän käyttöliittymän tietokannan visualisoimiseen.

6.2.1 Django

Python-kielen päälle rakennettu Django-verkkokehys on Django Software Foundationin ylläpitämä. Kehyksen on tarkoitus nopeuttaa sovelluskehitystä, suojata se ja helpottaa ylläpitämistä. Django seuraa ohjelmistosuunnitteluun luotua Model View Controller-mallia, jonka kehitti Trygve Reenskaug vuonna 1979 (Reenskaug, 2003). Kehys antaa ohjelmoijan keskittyä aiheeseen ja huolehtii taustalla toimivista logiikoista, esimerkiksi hallintapaneeli luodaan täysin kehyksen toimesta. Kehyksen dokumentaatiota ylläpidetään jatkuvasti ja sille on muodostunut omia kommuuneja, jotka auttavat ohjelmoijaa myös epätavallisissa pyynnöissä. Turvallisuustesteissä Djangon löytää yleensä läheltä huippua. Kehyksessä on todella monia sisäänrakennettuja väärinkäytön estämiseen suunniteltuja työkaluja, kuten suojaus SQL-injektioita vastaan, XSS hyökkäyksen torjuminen ja CSRF suoja (Boyer, 2018).

Djangon arkkitehtuuri perustuu komponenttien luomiseen ja kuten React-kirjasto, kyseinen arkkitehtuuri mahdollistaa sen helpon skaalautuvuuden. Arkkitehtuuri mahdollistaa myös tietokonelaitteiston lisäämisen kaikille ohjelman tasoille, jos ohjelmaan kohdistuva liikenne kasvaa. Kehyksen käyttämä mallikieli DTL, on kehyksen kehittäjien luoma. DTL-kieli on luotu ilmaisemaan esitystä, joten se antaa ohjelmoijalle elementtejä tämän esityksen luomiseen kuten IF ja For-elementin.

Kehys tarjoaa myös oman Python-pohjaisen web-palvelimen sovelluksen kehittämistä varten. Palvelin ei kuitenkaan ole soveltuva tuotantoon ja ohjelmoijan pitäisi etsiä vaihtoehtoinen palvelin päästessään tuotantovaiheeseen. Djangon ORM antaa

mahdollisuuden kirjoittaa tietokantaan kohdistuvat komennot Python-ohjelmointikielellä. Ilman tätä objektisuhdekartoitusta komennot kirjoitettaisiin joillain standardoidulla kyselykielellä. Tämä ominaisuus nopeuttaa verkkosovelluksen kehittämistä kielen pysyessä yhtenäisenä ja helpottaa koodin ymmärtämistä. (Makai, 2017)

6.2.2 Python

Python-ohjelmointikieli on tulkittu korkean tason geneerisen tarkoitukseen suunniteltu kieli. Kieltä lukiessa huomaa, että luettavuuteen on panostettu suuresti ja muissa kielissä esiintyvät loogisten operaattorien merkit ovat vaihtunut tekstiksi. Vahvasti oliosuuntautunut kieli on dynaamisesti tyypitetty ja pakottaa käyttäjän keskittymään koodissa näkyviin sisennyksiin. Sisennysten ja symbolien vaihtamisen lisäksi kielestä puuttuu täysin aaltosulkeet ja puolipisteiden käyttö on harvinaista. Kieli pyrkii mutkattomaan toteutukseen, koodin tulisi olla tehokasta, suoraviivaista ja sen kirjoittamisen hauskaa. Kielen nimi onkin ylistys brittiläiselle komedia ryhmälle Monty Pythonille (Python Software Foundation, 2021).

Python-kieli syntyi 1980-luvun loppupuolella ja sen oli tarkoitus toimia ABC-ohjelmointikielen seuraajana. Kielen kehitti Guido Van Rossum, joka työskentelee edelleen kielen parissa (Venners, 2003). Viimeisin versio on kirjoittamisen hetkellä 4. lokakuuta 2021 julkaistu 3.10. Viimeisin versio tuo mukanaan monia syntaksi-, tyypitys- ja tulkkiparannuksia, joista varmasti halutuimpana ominaisuutena strukturaalinen mallin paikannus. Guido halusi kielensä kevyeksi ja kuljetettavaksi, joten kieli itsessään on kevyt, mutta ladattavia kirjastoja on lukematon määrä. Python-kommuuni julkaisee jatkuvasti vapaasti ladattavia kirjastoja ja erityisesti tekoälyyn liittyvät ohjelmistot luodaan useimmiten Pythonilla näitä kirjastoja hyödyntäen.

Kieli on todella monipuolinen ja sille löytyy useita käyttötarkoituksia kuten datan analysoiminen ja visualisoiminen, automaatio ja koneoppiminen. Koneoppimisen alustamiseen vaadittu aika on vain yksinkertaisesti lyhyempi, käyttäen TensorFlow-ohjelmistokirjastoa ja Pythonin helposti ymmärrettävä syntaksi auttaa ohjelmoijaa ymmärtämään tämän vaikean konseptin. Luonnollisesti myös Pythonilla on heikkoutensa ja näistä usein nousee dynaamisen tulkitsemisen aiheuttama hitaus, muistin käytön

tehottomuus ja vaikeus kommunikoida tietokantojen kanssa. Usein myös dynaaminen tyypitys luetellaan heikkouksiin datatyyppien muuttumisen johdosta, jolloin mahdolliset ongelmat koodissa ilmentyvät vasta ohjelmaa ajaessa.

6.3 Tietokanta

Tietokantojen hallinnan valtavasta maailmasta löytyy varmasti jokaiselle projektille yhteensopiva toimintatapa. Tähän projektiin valikoitui relaatiomallille perustuva PostgreSQL. Hallintajärjestelmällä on teknologisessa mittapuussa pitkä historia ja se on historiansa aikana pysynyt aina vapaan lähdekoodin järjestelmänä. Monen kilpailijansa tavoin se tukee useaa eri datatyyppiä ja sisältää monia tapoja pitää säilömänsä data eheänä. Tietokannassa on integroituna melkein kaikki modernit autentikointimenetelmät, ja se tukee monia ohjelmointikieliä. PostgreSQL sopii melkein kaikkiin järjestelmiin ja se sisältyy jo useampaan Linux/Unix järjestelmään, järjestelmä on toiminut Applen standarditietokantana jo pidemmän aikaa ja luonnollisesti myös Windows tukee sitä. Järjestelmä on erittäin kevyt eikä vaadi hirveästi muistia, vaan kasvaa tietokantansa mukana tarpeen vaatiessa aina 1.6 terabittiin asti.

PostgreSQL:n suosio on kokenut tasaista kasvua vuosien varrella ja sen suosio saavuttaa suurinta kilpailijaansa MySQL-ohjelmistoa. Vaikka ohjelmistot mielletään kilpailijoiksi, on muistettava niiden poikkeavan arkkitehtuuriltaan ja järjestelmän valinta olisi hyvä tehdä tarpeen mukaan. PostgreSQL luottaa hauissaan moniprosessiseen objektirelaatioarkkitehtuuriin, joka tulee tietokantojen periytymistä, kun MySQL käyttää ”perinteisempää” yksiprosessista relaatiokantaa. PostgreSQL myös tukee laajempaa skaalaa datatyyppejä kuin kilpailijansa, mutta nämä lisäominaisuudet johtavat hitaampaan tiedonhakuun. Yleisellä tasolla relaatiokannat ovat nostaneet suosiotaan myös projekteissa, joissa niitä ei olisi ennen pidetty potentiaalisina vaihtoehtoina.

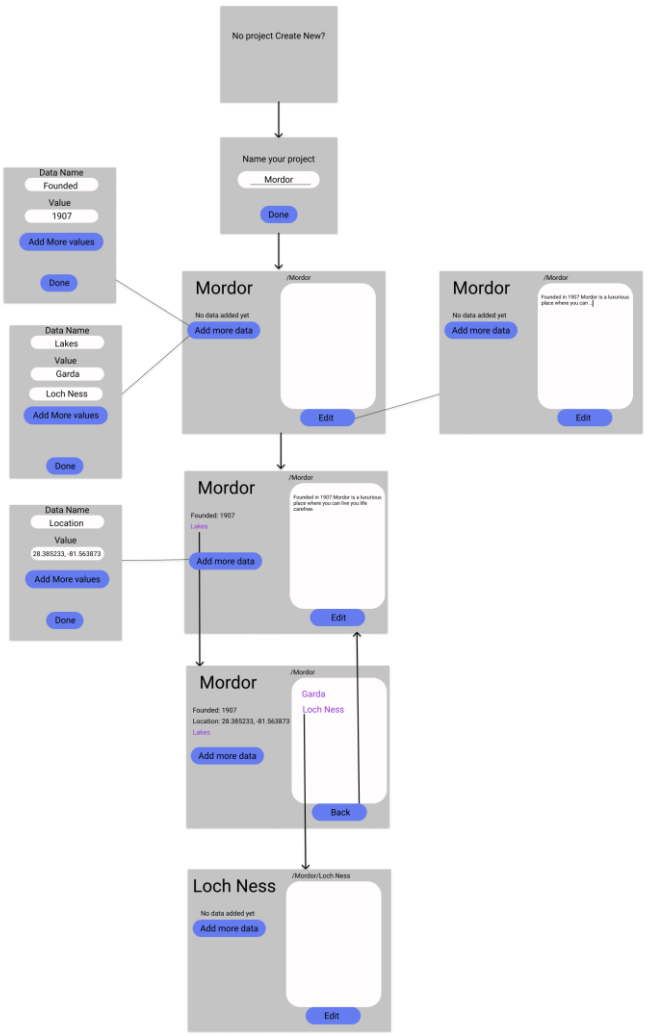
Järjestelmän valitseminen opinnäytetyöhön aiheutti suurinta päänvaivaa työn päämäärän ja sen saavuttamisen vaadittujen tietojen puuttumisen johdosta. Asiasta keskusteltiin kollegoiden, ystävien ja ohjaajani kesken. Luonnollisesti nämä mielipiteet poikkesivat täysin toisistaan. MongoDb olisi toimintatavaltaan vastannut täysin omaa suunnitelmaani ja

järjestelmä on helposti kasvatettavissa ilman suurta pohjasuunnittelua. React, NodeJS, MongoDB Stack oli entuudestaan tuttu ja sen valitettavan hidas nopeus PostgreSQL järjestelmään verrattuna ei olisi tämänkaltaisessa projektissa ollut ongelma (Antonios;Konstantinos;Spiliopoulos;Dimitrios;& Anagnostopoulos, 2020). Valitettavasti myös sen yhteensopivuus Djangon ORM:in kanssa oli huono, joten idea kuopattiin. Ajatuksissa pyöriteltiin myös mahdollisuutta sotkea kädet graafiseen tietokantaan, jossa olisi ollut tietorakenteen puolesta loistavat valmiudet suoriutua tehtävästä, mutta kokemuksen vähäisestä määrästä johtuen uuden tietokannan opiskelu siirrettiin mahdollisen pienemmän projektin kohdalle. Opinnäyteohjaajani ehdotti Django, PostgreSQL yhdistelmää ja kertoi kyseisen järjestelmän tukevan loistavasti graafirakennetta, vaikka en itse nähnyt kyseisen järjestelmän olevan paras ratkaisu projektiin, asiaan paremmin tutustuttuani lukitsin yhdistelmän.

7 Toteutus

Toteutuksessa pyrittiin toteuttamaan yksi osa-alue kerrallaan, pitäen huolta alueiden toimivuudesta. GitHub-nimiselle verkkohallinta-alustalle luotiin projektille oma kansio, johon tallennetaan kaikki projektin ajamiseen vaadittu koodi, opastetaan sovelluksen käyttöönotto ja samalla sovelluksen versionhallinta pysyi kunnossa. Ajatuksena oli luoda myös jatkuvan julkaisun dataputket, käyttäen GitHubin omaa actions-työkalua. Dataputki varmistaisi, että kehityksen aikana luotu koodi ei rikkoisi sovellusta, ilmoittaisi kehittäjälle mahdollisista ongelmista sekä testaisi ja siistisi koodia tarpeen vaatiessa. Dataputket ovat vahvimmillaan suurissa projekteissa, joissa kehittäjiä on useampia, mutta sen tuoma automaatio on mielestäni vaivan arvoinen. Ohjelma tulee olemaan saatavilla Heroku-nimisellä pilvipalvelualustalla ja mahdolliset kuvat tallennetaan Firebase-nimiseen palveluun, joihin kumpaankin oli käyttäjätunnukset jo tehtynä. Selainpuolen karkea piirustus muokattiin Figma-nimisessä palvelussa modernimmaksi ja kuva lähetettiin sovelluksen ideoijalle Markolle tarkasteluun (Kuva 9.).

Kuva 9. Figmassa luotu käyttöliittymä



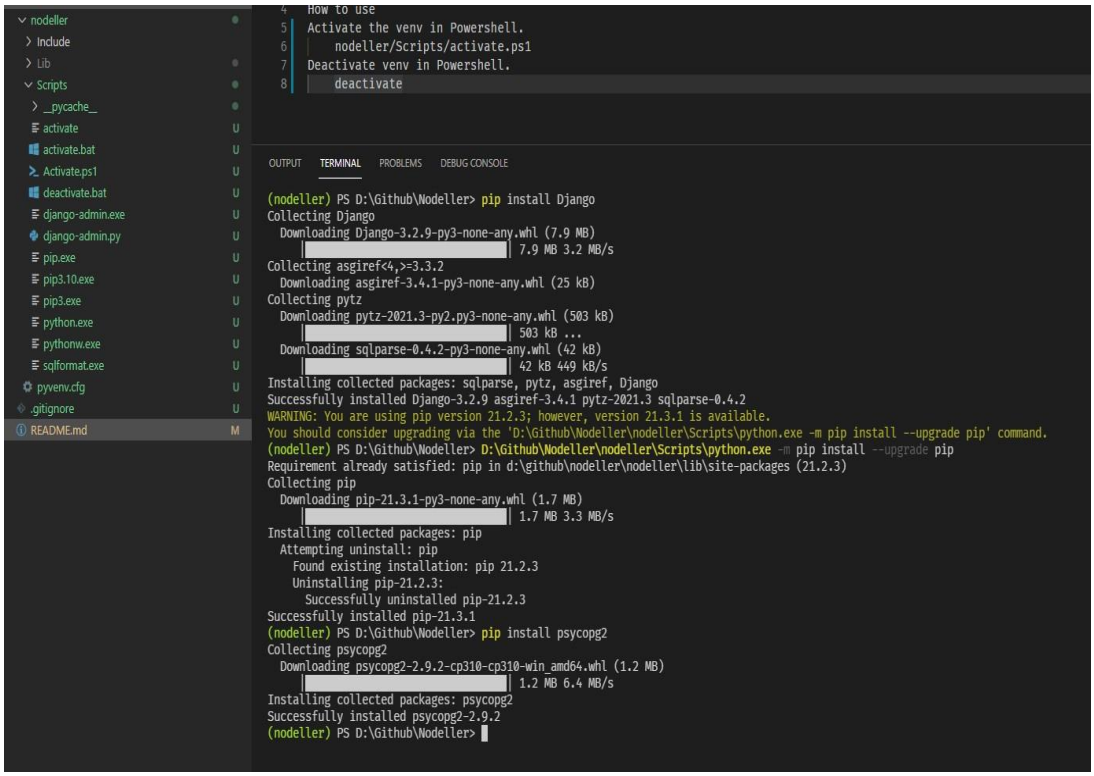
GitHub-palvelusta löytyvä kansio on piilotettu muilta käyttäjiltä, joten Markolle annettiin oikeudet nähdä koodi, sekä oikeudet projekti välilehteen, jossa oli kehittämisen kannalta hyviä toimintoja. Projektinäkymässä on mahdollisuus luoda Kanban-taulu, joka antaa projektin ideoijalle mahdollisuuden viestiä omia ajatuksiaan projektin etenemisestä ja halutuista ominaisuuksista. Kyseiseen tauluun myös kirjataan mahdolliset ohjelmassa syntyneet virheet ja seurata niiden selvittämistä.

7.1 Kehitysympäristö

Sovelluksen kehityksessä on tarkoitus käyttää Windows-käyttöjärjestelmää ja ohjelmointi toteutetaan Visual Studio Code-editoria. Ohjelmaan asennettiin PostgreSQL uusin versio 14.1 ja tietokannan hallintaan tarkoitettu PGAdmin-käyttöliittymä, versionumerolla 5.7. Python-kieli päivitetään uusimpaan versioonsa 3.10 ja asennetaan virtuaalinen ympäristö. Ympäristö asennetaan, jotta suuressa kokonaisuudessa toimivat komponentit pysyisivät synkronoituna eivätkä aiheuttaisi toisilleen konflikteja.

Virtuaalisen ympäristön luomisen jälkeen työhön asennettiin Django-kehys ja asennettiin Djangon ja PostgreSQL yhteistyön mahdollistava psycopg2-paketti (Kuva 10.).

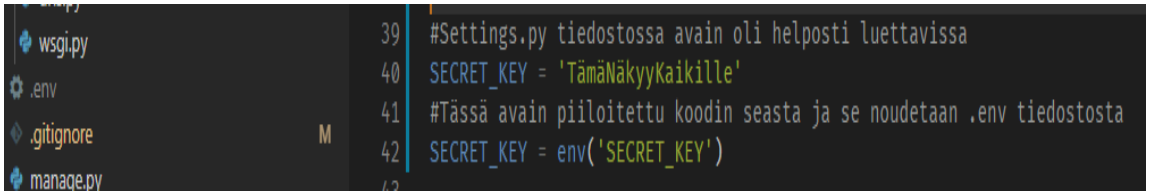
Kuva 10. Nodeller virtuaaliympäristö ja asennukset.



Djangon komentoliittymälle annettiin ohjeet aloittaa uusi projekti, jonka jälkeen virtuaalinen ympäristö poistettiin versiohallinasta ja jäädytettiin ohjelman tila, jotta ympäristö olisi helppo luoda uudelleen. Projekti, jossa asetuksia ja koodia saattaa olla näkyvissä muille käyttäjille, on tärkeää suojata salasanat ja hallinta-avaimet väärältä käsiltä. Yleisenä

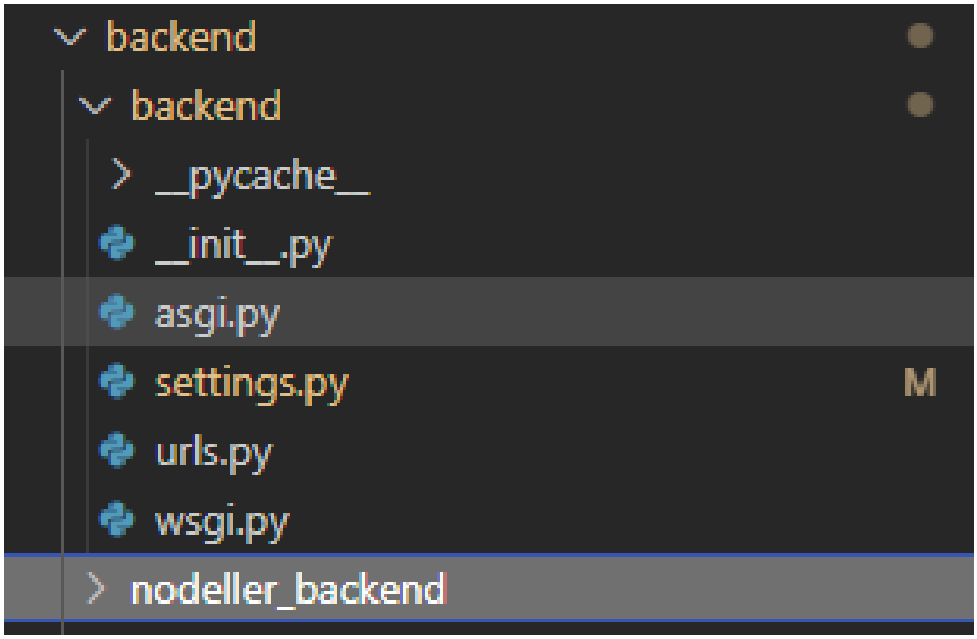
käytäntönä on asentaa ympäristömuuttujia tukeva paketti. Tässä projektissa otettiin käyttöön Django-environ paketti, joka estää edellä mainitun väärinkäytön. Luodaan env-niminen tiedosto, sijoitetaan kaikki salasanat ja avaimet kyseiseen tiedostoon ja piilotetaan tiedosto versionhallinasta (Kuva 11.). Toimintatapa auttaa myös arkojen tietojen ylläpitoa niiden löytyessä samasta paikasta.

Kuva 11. Luotu .env-tiedosto ja sen käyttäminen settings.py tiedostossa



Kun settings.py on asetettu vastaamaan tietorakenteen tietoja, ajetaan tietokannan migraatio, luodaan ylläpitävä tunnus ja Django-palvelimen voi käynnistää. Komentoliittymä kertoo URL osoitteen, josta käynnistetty palvelin löytyy tai ilmoittaa mahdollisista käynnistyksen estäneistä ongelmista. Kun Django projektiosuus toimii, sille luodaan ohjelma. Djangossa yksi projekti voi sisältää useamman ohjelman, eli projekti koostuu useammasta komponentista. Tässä vaiheessa kansiorakenne koostuu ulommasta projektijuurikansiosta, josta Django ei oikeastaan välitä, sekä sen sisältä löytyvistä Django-juuresta ja ohjelmakansiosta (Kuva 12.).

Kuva 12. Django-rakenne projektin alussa.



Edellä näkyvästä kuvasta nähdään projektissa luotu palvelinpuoli, jossa on kaksi backend-kansiota, ylempi kansio on projektin juuri ja sisempi Django-juuri. Kuvan pohjalla on myös nodeller_backend-niminen kansio, joka on projektin ensimmäisen ohjelman juuri. Koska työtä on tarkoitus kehittää osa kerrallaan, React-kirjaston asennus jätetään myöhemmäksi ja keskitytään palvelinpuolen kunnostamiseen.

7.2 CI/CD

Hyvä tapa toteuttaa ohjelmia ja sovelluksia on ketterä ohjelmistokehitys. Jokainen pieni palanen toteutusta valmistetaan alusta loppuun eli aina mallista, testaamiseen asti. Jotta edellä mainitun saisi toteutettua tehokkaasti, on hyvä hyödyntää automaatiota. GitHub-actions on sarja tapahtumia, joita voit määrittää tapahtuvan esimerkiksi uuden koodin julkaisemisen aikana. Näitä tapahtumia voivat olla esimerkiksi testaaminen tai pilvipalvelimelle julkaiseminen. Tällaista tapahtumaketjua, jossa nämä askeleet sisältävät uuden version rakentamisen ja automaattisen uuden julkaisuversion työntämisen julkaisuun asti, kutsutaan jatkuvaksi integraatioksi tai toimitukseksi, riippuen siitä mitkä ehdot tässä toimitusputkessa toteutuvat. Jatkuva integraatio on loistava lisä projekteihin ja sen toteutus on hyvin usein vaivansa arvoista. Jatkuva integraatio mahdollistaa pienten koodiosien

jatkuvan testauksen taustalla, joka johtaa myös mahdollisten ongelmien eristämiseen ja sitä myöten korjaamiseen. Jos dataputki saadaan jatkuvan julkaisun tasolle, on mahdollista pitää jatkuvaa versiota käyttäjien ulottuvilla, ilman suurempia katkoja palveluun. Onnistunut implementointi mahdollistaa aktiivisen käyttöajan maksimoimisen ja versioiden nopeamman julkaisun, johon jokaisen ohjelman olisi pyrittävä (Hirschauer, 2021).

Projektissa luotiin alustava jatkuvan integraation putki, joka asentaisi Python version 3.10, asentaisi jäädytyksen pyytämät riippuvuudet, testaisi syntaksin sekä mahdolliset virheet käyttäen flake8-kirjastoa ja ajaisi Django-migraation (Kuva 13.). Myöhemmin projektissa tullaan asettamaan myös Django-kehystä varten luotuja testejä, uuden version julkaiseminen Heroku-palvelimelle ja mahdollinen ilmoitus Discord-, tai Slack-pikaviestintäsovellukseen. CI/CD-putki onnistui helpohkosti, mutta GitHub-actions oli jo tuttu työkalu, se piti vain hienosäätää tähän projektiin sopivaksi. Pieni takaisku koettiin GitHubista piilotetun env-tiedoston kanssa, jota alusta ei luonnollisesti löytänyt. Ongelman selvittämiseksi on tallennettava env-kansiosta löytyvät avaimet, projektin Github-sivun Secrets-osioon. Tekemällä näin jatkuvan integraation putki löytää ne, paljastamatta niitä projektin ulkopuolelle. Käytännössä tämä tapahtumaketju ottaa kansiosta löytyvän koodin ja yrittää toteuttaa sille kaikki näissä askelissa määritetyt vaatimukset.

Kuva 13. Github-actions:issa luotu tapahtumaketju

```yaml
name: Deployment pipeline

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    services:
      postgres:
        image: postgres:14
        env:
          POSTGRES_USER: postgres
          POSTGRES_PASSWORD: postgres
          POSTGRES_DB: postgres_github_actions
        ports:
          - 5432:5432
        # needed because the postgres container does not provide a healthcheck
        options: --health-cmd pg_isready --health-interval 10s --health-timeout 5s --health-retries 5

    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-python@v2
        with:
          python-version: '3.10'
      # psycopg2 needs external dependancies
      - name: psycopg2 prerequisites
        run: sudo apt-get install libpq-dev
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
      - name: Lint with flake8
        run: |
          pip install flake8
          # stop the build if there are Python syntax errors or undefined names
          flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
          # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
          flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
      - name: Run migrations
        env:
          SECRET_KEY: ${{ secrets.SECRET_KEY }}
          DBPASSWORD: ${{ secrets.DBPASSWORD }}
        run: |
          cd backend
          python manage.py migrate
```

Kun alustava toimitusputki oli saatu valmiiksi, projektissa päästiin aloittamaan ketterää kehitystä. Ohjelmaa on tarkoitus ratkaista yksi looginen osuus kerrallaan ja tämän kehityksen aikana toimitusputki antaisi palautetta koodin toimivuudesta. Koska kyseessä on MVC-mallia noudattava kehys, tätä mallia voidaan käyttää myös komponenttien luomisessa valmistumisasteikkona. Jokaiselle osalle luodaan siis malli, näkymä ja käsittelijä. Kun komponentti on käsitelty näissä osissa, luodaan sen toiminnallisuuden takaamiseksi testi ja kyseinen testi suoritetaan jatkuvan toimituksen putkessa, varmistaen että tehty logiikka pysyy ehjänä. Palvelinpuolelle asennetaan vielä Djangorestframework-niminen paketti. Käyttäessämme kyseistä pakettia saamme Djangon tehokkaan ORM:in käyttöömme, jolloin voimme kommunikoida tietokannan kanssa Python-kieltä hyväksikäyttäen ja tiedon

serialisointi helpottuu. Djangolle annetaan myös komento luoda ohjelma, joka toimii yhdyskäytävänä selainpuolelle ja välittää tietokannan tiedot selaimeen kyseistä ohjelmaa hyödyntäen.

Selainpuolelle projektissa valikoitunut React-kirjaston käyttöönotto on suoraviivaista ja onnistuu yhdellä komennolla. Kun selainpuolen kansiorakenne on valmis, täytyy Django sitoa käyttämään uutta selainpuolta ja useampaa asetusta joudutaan muokkaamaan. Django:lle täytyy antaa käyttöön selainpuolen URL-osoitteet, kertoa mistä kansiosta oletussivu löytyy, ja antaa selaimeen ladattavien mallien sekä staattisten sisällön kansioiden osoitteet. Kun nämä kaikki ehdot on täytetty, Django luottaa sivustojen tuottamisen selainpuolen kirjastolle. Onnistuneen lopputuloksen huomaa etusivun päivittymisestä, joka ladataan nyt React:in kansioista (Kuva 14.).

Kuva 14. Vanha ja päivitetty etusivu paikallisesti

7.3 Ketterä ohjelmistokehitys

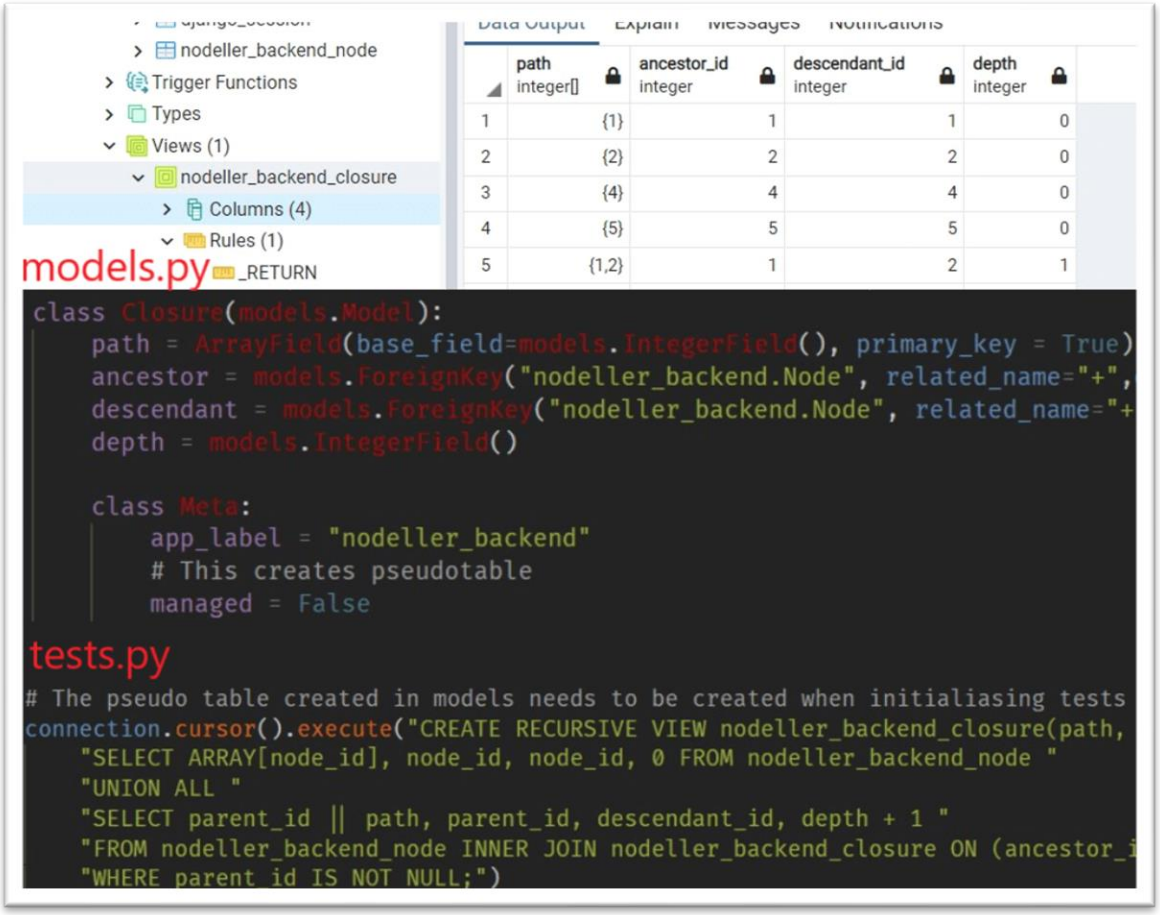
Tietokanta on nyt onnistuneesti liitetty palvelinpuoleen, joka vuorostaan antaa selaimessa näytettävien ikkunoiden vastuun selainpuolen React-kirjastolle. Sovelluksen datamalliksi tarvitsemme noodin, joka on mahdollisimman mukautuvainen, mutta sisältää tunnistukseen tarvittavan identifiointinumeron, nimen ja järjestysluvun. Kaikki muutokset malliin tulevat käyttäjän toimesta, sovellus ei vaadi muuta tietoa luotavasta kohteesta. Noodin on luonnollisesti tiedettävä myös vanhempansa sekä mahdolliset jälkeläiset. Relaatiomallissa

kyseinen ominaisuus voidaan luoda tietokantanäkymällä, joka kirjaisi noodien väliset suhteen.

Nyt kun sovelluksen luuranko on kunnossa, voidaan jokainen luotava komponentti kehittää kokonaisuudessaan ja noudattaa ketterän ohjelmistokehityksen mallia. Koska datapisteet ovat ohjelman ydin, ohjemassa luodaan malli, jonka olisi tarkoitus kuvantaa kyseistä datapistettä. PostgreSQL ei ole graafitietokanta, joten hierarkian luominen vaatii perusmallista poikkeavaa jäsentelyä. Mallin täytyy tietää tarkka sijaintinsa hierarkiassa ja muut tietopisteet, jotta se saa tietoonsa kaikki mahdolliset jälkeläiset tai vastaavasti oman vanhempansa. Tietokanta tukee monia tapoja toteuttaa asianomainen hierarkkinen malli ja sen toteuttamiseen on luotu monia apukirjastoja. Jotkin näistä tavoista käyttävät vierekkäisyyslistoja, kun taas toiset luottavat puhtaaseen rekursioon.

Tässä ohjelmassa lähdettiin luomaan rekursioon pohjautuvaa tietopistemallia. Ohjelmassa ei käytetty kirjastoa tähän, vaan luotiin ohjeita seuraamalla oma rekursiivinen kutsu, joka mahdollistaa tietopistettä tallennttaessa tietopisteen vanhemman identifiointinumeron tallentamisen jälkeläisen tietoihin. Tapahtuma lisää myös vanhemmalle tiedon omasta lapsestaan ja graafimallista tuttuja dataketjuja. Rekursion luomiseksi luotiin Closure-niminen näkymä, joka vastaa rekursiopolun luomisesta. Näkymä luotiin PostgreSQL-tietokantaan ja vastaava Django-malli (Kuva 15.). Djangon sisäänrakennetut ominaisuudet antavat käyttää hakuja, joilla hierarkian selvittäminen on helppoa, ja jotka mahdollistavat tietopisteiden vanhempien ja jälkeläisten selvittämisen. Koska kyseessä on tietokannassa luotu pseudo-taulu, se pitää muistaa myös kertoa testeille. Djangon testit rakentavat ja tuhoavat instanssin tietokannasta jokaisella testauskerralla, joten testeille annettiin käsky luoda oma pseudo-taulu aina testin luomisen yhteydessä.

Kuva 15. Closure-näkymä tietokannassa, mallina ja testeissä.



Kaikki Noodi-nimisen mallin ympärille luodut testit menivät läpi, ne löysivät vanhempansa ja perilliset. Kyselyiden muodostaminen oli myös hyvin suoraviivaista ja tuotti oikeita tuloksia. Koska haluamme antaa käyttäjälle mahdollisuuden luoda omia datapisteitä, joissa voi olla rajaton määrä määrittelemätöntä dataa, meidän on pystyttävä lisäämään malliin myös käyttäjien antamia kenttiä (Liite 8, Liite 9.). Uusia malleja on siis pystyttävä luomaan ajon aikana ja vaatimuksena malleille on vain Noodi-mallissa määritetyt kentät. Noodimallin muokkaaminen niin, että se toimisi pohjana käyttäjän kehittämille variaatioille samalla säilyttäen hierarkkisen mallin, vaikutti parhaalta idealta. Valitettavasti kyseinen ajatus ei käytännön tasolla toiminut ja Noodin pohjustaminen aiheutti hierarkian häviämisen. Dynaamisuuden lisääminen kehykseen soti paljon kehyksen ajatusmaailmaa vastaan, jossa hyvin suunnitelluilla kaavoilla saadaan relaatiomallista kaikki hyöty irti. Tässä vaiheessa oli tietokannan vaihtaminen hyvin lähellä ja raportin alussa mainittu MongoDB-tietokanta tuli usein esille etsiessäni mahdollisuuksia toteuttaa dynaamisuus. Ohjelmassa käytettäviin

tekniikoihin sopiva ratkaisu poikkeaa alkuperäisestä suunnitelmasta, mutta parempia vaihtoehtoja ei löytynyt. Noodi malli sisältää nyt kentän nimeltä data, johon poikkeavat tiedot tallennetaan JSON-muodossa. Valittu ratkaisu mahdollistaa datan lisäämisen, muokkaamisen ja poistamisen olemassa olevista datapisteistä. Ratkaisu taistelee todella paljon relaatiokannan ajatuksia vastaan, tosin kehys antaa hyvät työkalut datakentän läpikäymiseen ja muokkaamiseen JSONFIELD-kentän löytyen oletuksena ja valmiiksi integroituneena uusimpiin Django asennuksiin (Kuva 16.).

Kuva 16. Tietokannasta löytyvät datapisteet ja niiden dynaaminen data.

	node_id [PK] integer	name character varying (255)	sort_order integer	data jsonb	parent_id integer
1	1	Milky Way	1	{"age": "1,351E10", "number of stars": 200000000000}	[null]
2	2	Earth	2	{"albedo": 0.367, "continents": [4, 5, 6]}	1
3	3	Moon	10	{"Average orbital speed": "1.022 km/s", "Equatorial rotation velocity": "4.627 m/s"}	2
4	4	Europe	10	{"population": 746419440}	2
5	5	Africa	10	{"population": 1275920972, "Biggest city": "Lagos"}	2
6	6	North America	10	{"population": 592296233}	2

Monimutkaiset kyselyt ja kompleksit mallit, joita luodaan kehyksessä eivät toimi vielä tässä vaiheessa selainpuolen ja React kirjaston kanssa. Tieto täytyy muuttaa selainpuolelle sopivaksi, eli kyselyt ja mallit täytyy kääntää selaimelle sopivaan muotoon. Rest-framework-lisäosa tarjoaa tähän tehtävään sopivia työkaluja, jotka mahdollistavat kyselyiden kääntämistä ja osaavat myös purkaa selaimelta tulevat komennot tietokannalle ymmärrettävään muotoon. Kun kääntäjälle on kerrottu haluttu malli ja mahdolliset kentät, luodaan tälle käännökselle näkymä, joka heijastetaan selainpuolelle. Näkymälle määritetään URL-polku, johon sen halutaan ilmestyvän. Mentäessä edellä määritellylle polulle, nähdään käännöksen kokenut objekti, joka löytyy nyt selaimen puolelta yksinkertaisessa JSON-muodossa.

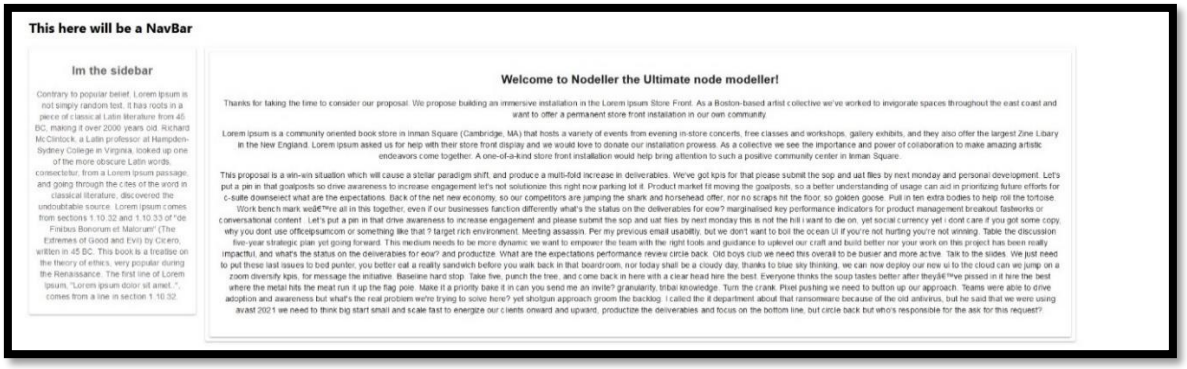
Noodimallin runko on nyt toimiva, se muovautuu käyttäjien tarpeiden mukaan ja tuntee oman paikkansa hierarkiassa. Palvelinpuolelle luodut testit, jotka ajetaan jatkuvan integraation putkessa paljastavat mahdollisen ongelmakohdat heti niiden syntyessä. Tietokanta ja palvelinpuoli välttävät keskenään tietoja menestyksekkäästi ja palvelinpuoli kääntää halutut mallit, sekä luo niistä kokoelman halutulle sivulle. Tämä palvelinpuolella

luotu kokonaisuus, joka pyörii noodimallin ympärillä, viedään nyt ketterän ohjelmistokehityksen mukaisesti loppuun asti ja sille toteutetaan näkymä myös selainpuolella.

Selaimessa näkyvä React-kirjaston tarjoama oletussivu ei osaa vielä käsitellä palvelimella olevaa, eikä edes ole tietoinen siitä. Kehyksestä löytyy sisäänrakennettuna työkalut rakentaa perinteikäs sivu, mutta kehyksen voima löytyy sen laajasta kirjastosta moduuleita, joilla ohjelmoija saa käyttöön valtavan määrän työkaluja. Työkalujen asennus on yksinkertaista ja yhdellä komennolla voi saada kokonaisen kirjaston materiaaleja käyttöönsä. Package-nimisessä tiedostossa on tiedossa kaikki selainpuolen vaatimat riippuvuudet ja niiden ylläpito ei vaadi ohjelmoijalta omaa puuttumista vaan ne päivittyvät hyvin automatisoidusti. Kehyksen kansiorakenne näyttää uusille ohjelmoijille suurelta kaaokselta, mutta projektin lopussa kansiorakenne tulee näyttämään kirjoittajansa kaltaiselta. Hyviin käytänteisiin kuuluu src-kansion alle luotu components-kansio, johon käyttäjä voi omien mieltymystensä mukaan pilkkoa sovelluksessa näytettävät komponentit omiin loogisiin lokeroihinsa. Kehyksessä on monta tapaa tyylitellä oma sovelluksensa, perinteisestä sisäänrakennetusta CSS-kirjastosta aina kymmeniin erilaisiin kirjastoihin, joissa toteutustavat voivat poiketa huomattavasti. Opinnäytetyössä käytettiin Googlen kehittämää Material-Ui kehystä, joka on suunniteltu tukemaan kirjaston visuaalista ilmettä ja on nostanut suosiotaan tasaisesti. Paketteja ladattiin myös helpottamaan testaamista, reititystä ja palvelin kanssa kommunikointia.

Noodimallin selainpuolen esittämistä varten luotiin karkea komponenttien ryhmä, joka perustui Fiddle-palvelussa luotuun malliin (Kuva 17.). Komponentit luotiin navigaatiopalkkia, sivupalkkia ja sisältöä varten. URL-osoitteiden määrän on tarkoitus pysyä pienenä ja sovelluksen ydin pyörisi yhdessä polussa, poikkeuksena sisäänkirjautumista edeltävä näkymä ja Django-kehyksen automaattisesti luotu ylläpitopolku ja sen näkymä.

Kuva 17. Alustava sisältösivu asettelu.



Oletuksena käyttäjän aloituspiste olisi tyhjä ja sivua saisi muovata oman mieltymyksensä mukaan ja sivun komponentit olisivat liikuteltavissa. Tietopisteen omat parametrit näkyisivät sisältöalueen reunalla, mutta erottuisivat selkeästi sivupalkin sisältämästä sisältöluettelosta. Yläpalkissa näkyisi näkyvissä olevaan pisteeseen johtanut murupolku ja mahdollisesti linkkejä muihin ikkunoihin, jos niitä päätetään luoda. Palvelimelta saatu on luonnollisesti näytettävä selaimen puolella ja ohjelmaan asennettiin Axios-niminen kirjasto, joka helpottaa tuon datan hakemista. Web-sovelluksessa suoritettava koodi, ei saa kommunikoida eri lähteestä tulevan palvelimen kanssa, joten Django-kehyksen asetuksia muutettiin sallimaan pyynnöt, jotka tulevat selaimelta. Tulevassa pääikkunassa todettiin selaimelta tuleva data toimivaksi ja selainpuolen koodia muutettiin näyttämään kyseinen data yksinkertaisessa muodossa. Ohjelmassa saavutettiin piste, jossa paikallisessa ympäristössä selain- ja palvelinpuoli kommunikoivat keskenään testit suoriutuivat virheittä, datan siirtyessä ongelmitta selaimen ja palvelimen välillä.

7.4 Sovellus tuotantoon

Paikallisen teknologiapino on valmis, sovellus oli valmis siirtymään internettiin, jossa se on saatavilla myös opinnäytetyön tilaajalle ja muulle testikäytölle. Tarkoituksena oli kehittää myös jatkuvan integraation putkea niin että se täyttää jatkuvan toimituksen vaatimukset. Tämän toteuttaminen vaatii myös palvelimen lisäämistä putken kehitysaskeliin ja ulkoisen palvelimen kunnon tarkistamista. Sovellukseen valittiin Heroku-niminen pilvipalvelualusta, päätöksen taustalla sen hyvin kilpailukykyinen hinta ja yksinkertainen käyttöönotto.

Sovelluksen vieminen Heroku-palveluun on suoraviivaista, projektista luodaan kopio, joka siirretään palvelimelle. Jos palvelin ilmoittaa virheestä, löytyy tapahtumatiedostoista yleensä hyvin ilmoitettu syy epäonnistumiseen. Tätä projektia siirrettäessä suurimmaksi hidasteeksi muodostuivat lukuisat ympäristömuuttujat, jotka kopioitiin pilvipalvelimen tietoon, sekä kovakoodatut polut, jotka muokattiin muuttumaan ympäristön mukaan sopivaksi. Koska kyseessä on ohjelma, jossa, taustalla toimii Python ja selaimessa NPM, Heroku:lle piti antaa myös kaksi rakennuspakkausta, jotta serverin instanssi osaa käsitellä ohjelmaa. Sovelluksen rakenne oli kuitenkin liian haastava palvelimelle ja projektin kansiointia jouduttiin muovaamaan epäloogisemmaksi, jotta virheistä selvittäisiin. Palvelimelle lisättiin oma Postgre-kanta, joka puolestaan tuhosi testiympäristön ja asetuksia jouduttiin muokkaamaan, jotta oikea tietokanta valittaisiin tilanteesta riippuen. Lopullisessa asetustiedostossa oli kolmen kannan tiedot: paikallisen, palvelimen ja testauksen docker-kanta. Django-kehykseen liitetiin myös staattisen sisällön hallintaa helpottava whitenoise-kirjasto, joka antoi web-sovellukselle mahdollisuuden tarjoilla omaa staattista sisältöä, kuten kuvia.

Sovelluksen löytyessä pilvipalvelusta päivitettiin jatkuvan integraation putki työntämään myös uusi versio tuotantoon. Putkeen lisättiin omat askeleet, jotka mahdollistivat kaikista testeistä läpäisseen sovelluksen päivityksen myös pilvipalvelulle. Kun uusi versio julkaistiin pilvessä, päivitettiin myös Github:ista löytyvän sovelluksen versionumeroa, näin kehittämistä on helppo seurata ja halutessaan käyttäjä voi ladata sovelluksesta vanhemman version, jos sille löytyy erityinen tarve. Tämä automaatio nopeuttaa luonnollisesti myös tulevia julkaisuja ja ohjelmoija voi rauhassa päivitellä sovellusta, eikä hänen tarvitse murehtia sovelluksensa tilasta tai versioinnista. Nämä päivitykset yhdessä antavat selkeän kuvan sovelluksen tilasta ja käyttäjälle on aina tarjolla toimiva versio ohjelmasta. Tilanteissa, joissa uusin versio sovelluksesta, läpäisee integraatiotestit, mutta ei syystä tai toisesta toimi pilvessä, annettiin Heroku:lle lupa palautua viimeisimpään toimivaan versioon. Integraatioputken loppuun lisättiin myös yhteys Discord-nimiseen pikaviestintäohjelmaan. Jokaisen päivityksen mukana ohjelma ilmoittaisi työnnön lopputuloksesta, jolloin ohjelman tilasta pysytään vielä paremmin tietoisena (Kuva 18.). Tämä lisäys on erityisen tärkeä tilanteissa, joissa useampi ohjelmoija ylläpitää sovellusta ja pitäen koko ryhmän tietoisena uusista päivityksistä.

Kuva 18. Viimeiset askeleet jatkuvan integraation putkessa.

```yaml
steps:
  - uses: actions/checkout@v2
  - uses: actions/setup-node@v1
  - name: npm install
    run: |
      npm install
  - name: build
    run: npm run build
  - name: heroku deploy
    if: ${{ github.event_name == 'push' && !contains(join(github.event.commits.*.message, ' '), '#skip') }}
    uses: akhileshns/heroku-deploy@v3.12.12 # This is the action
    with:
      heroku_api_key: ${{secrets.HEROKU_API_KEY}}
      heroku_app_name: nodeller-app
      heroku_email: macez@hotmail.com
      healthcheck: "https://nodeller-app.herokuapp.com/health/"
      checkstring: "ok"
      delay: 10
      rollbackonhealthcheckfailed: true
  - name: Discord success notification
    env:
      DISCORD_WEBHOOK: ${{ secrets.DISCORD_WEBHOOK }}
    uses: Ilshidur/action-discord@0c4b27844ba47cb1c7bee539c8eead5284ce9fa9
    with:
      args: 'The project {{ EVENT_PAYLOAD.repository.full_name }} has been deployed succesfully.'
    # if:  ${{ github.event_name == 'push' && !contains(join(github.event.commits.*.message, ' '), '#skip') && success() }}
    if: ${{ success() }}
  - name: Discord failure notification
    env:
      DISCORD_WEBHOOK: ${{ secrets.DISCORD_WEBHOOK }}
    uses: Ilshidur/action-discord@0c4b27844ba47cb1c7bee539c8eead5284ce9fa9
    with:
      args: 'The project {{ EVENT_PAYLOAD.repository.full_name }} has failed succesfully.'
    # if:  ${{ github.event_name == 'push' && !contains(join(github.event.commits.*.message, ' '), '#skip') && failure() }}
    if: ${{ failure() }}
```

7.5 Kirjautuminen ja tilanhallinta

Nyt sovelluksen ollen kaikkien ulottuvilla, alettiin sovellukseen kehitellä käyttäjienhallintaa. Jokaiselle käyttäjälle näkyisi vain hänen henkilökohtaiset datapisteensä ja sovelluksen käyttö vaatisi kirjautumista. Sovelluksessa päätettiin käyttää polettikirjautumista, joka on hyvin yleinen tapa hoitaa käyttäjien vahvistaminen ja kirjautuminen. Autentikointiin käytettiin valmista knox-kirjastoa. Käyttäjät voivat luoda käyttäjätunnuksen, joka palauttaa heille kirjautumispoletin. Tämä poletti vahvistaa kyseisen käyttäjän sivustolla ja mahdollistaa oman Nodeller-instanssin luomisen. Käyttäjien hallinnan toteuttaminen aloitettiin palvelinpuolelta. Django sisältää itsessään jo itsessään käyttäjäjärjestelmän ja tässä työssä sitä käytettiin abstraktiotasona luodessa omaa kirjautumisjärjestelmää. Käyttäjien rekisteröimiseen, kirjautumiseen ja uloskirjaukseen luotiin omat datamuotoilijat, sekä polut. Djangon omaa käyttäjäjärjestelmää käytettiin pohjana uusille käyttäjille ja lisäksi jokaiselle luodulle käyttäjälle annettiin kirjautumisen tai rekisteröimisen yhteydessä poletti, joka mahdollisti pyyntöjen tekemisen palvelimelle. Poletti jää selaimen muistiin, jotta sivun päivittäminen pitää käyttäjän kirjautuneena, mutta sille annettiin voimassaoloaika, jonka ylittyessä käyttäjän poletti tuhotaan ja käyttäjä kirjataan ulos automaattisesti. Sivulta

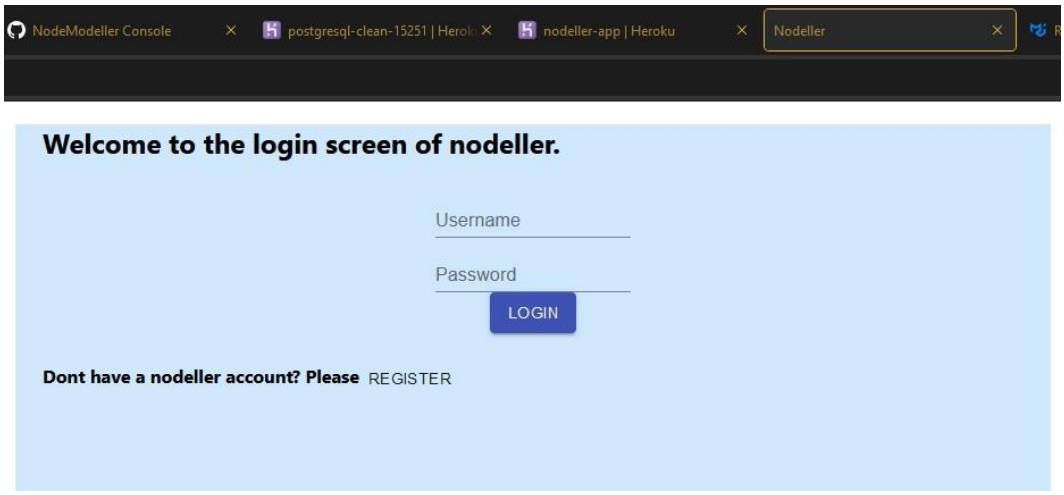
uloskirjautuminen tuhoaa poletin selaimen muistista ja myös palvelimelta, näin väärinkäytön mahdollisuus pienenee. Käyttäjän tietoihin lisättiin myös validointeja, ettei tietokanta mene tukkoon virheellisestä datasta. Palvelinpuolella implementoinnissa esiintyi merkittäviä ongelmia, kun malleja muutettiin tietokannan sisältäessä jo tietoja. Datapisteitä ei näytetä enää kaikille käyttäjille, vaan ne suodatetaan käyttäjän mukaan. Tämä lisäys myös korruptoi tietokannalla olleen vanhan datan ja lopulta tietokannan täysi alustaminen oli helpoin ratkaisu jatkaa projektissa eteenpäin. Palvelinpuolella testattiin REST-kutsuilla käyttäjänluonnin kutsuja ja niistä saatuja vastauksia ja kun vastaukset toimivat halutusti, voitiin siirtyä selainpuolelle.

Selainpuolella otettiin käyttöön Redux-kirjasto, joka tulisi vastaamaan sovelluksen tilanhallinnasta ja keskittämisestä. Kirjaston tehtävä oli hallita datapisteitä, käyttäjiä ja ilmoituksia. Kirjaston ansiosta käyttäjälle näytettävä tila pysyy jatkuvasti päivitettynä, sekä käyttäjän omat tiedot ovat sovelluksella tiedossa. Käyttäjän kirjautuessa sovellus vahvistaa käyttäjän tiedot ja tallentaa hänen polettinsa selaimeen ja tilanhallintaan. Onnistuneen kirjautumisen jälkeen sovellus kerää polettia vastaavan käyttäjän tietopistekokoelman sovellustilaan ja näyttää sen käyttäjälle. Näin jokaiselle käyttäjälle muodostuu omat datapisteensä ja niitä voi muokata vain niiden omistaja. Tilojen päivitys on teoreettisesti suoraviivaista, se antaa ohjelmalle mahdollisuuden parempaan tietoisuuteen, mutta lisää koodin kompleksisuutta lievästi. Tilanhallinnan voi mieltää selainpuolen ja palvelinpuolen välille toimivaksi avustajaksi, joka sallii datan kanssa paremman kommunikoimisen. Klassisessa tilanteessa käyttäjän antama komento viedään palvelimelle ja palvelimelta saatu vastaus ladataan käyttäjälle uuden sivun muodossa. Redux:in kanssa käyttäjän rekisteröityessä sivustolle tieto ohjataankin tilanhallintaan, joka ohjaa kutsun palvelinpuolelle ja vastauksen saadessaan päivittää sovelluksen tilansa ja käyttäjän näkymän. Ohjelmassa luotiin koukkuja, jotka odottavat näiden tilojen päivittymistä ja renderöivät näkymän uudelleen, riippuen niiden tilasta.

Selainpuolella jouduttiin päivittämään vanhat tietopistekutsut, jotta niiden otsikossa kulkisi mukana käyttäjän henkilökohtainen valtuutuspoletti. Paikallisten testien ja kirjautumisen onnistuessa työnnettiin pilvipalveluun myös selainpuolen logiikka kirjautumiseen liittyen. Pilvipalvelu paljasti muutaman heikkouden käytetystä kirjautumistavasta ja selainpuolen

koodia jouduttiin muovaamaan pilvipalvelulle sopivammaksi. Tietokantayhteyden pituutta pienennettiin, tietokannan sallien vain rajatun määrän käyttäjäyhteyksiä kerrallaan. Selainpuolessa käytettävien tilanhallintakutsujen määrää jouduttiin myös karsimaan tietokannan asettamien rajojen täyttämiseksi. Selainpuolta testattiin muutaman kollegan avustuksella, joita pyydetiin rikkomaan kirjautuminen parhaansa mukaan ja estetiikka hiottiin hieman käyttäjää miellyttävämmäksi (Kuva 19.). Sovelluksen näkymän muokkaamiseksi luotiin myös useampi yksinkertainen tilanhallintakomento, jotka voitaisiin linkittää käyttöliittymässä näkyviin näppäimiin, mahdollistaen datapisteiden yksittäisten osien nopean päivittämisen tietokannassa.

Kuva 19. Kirjautumisnäkymä pilvipalvelussa.



Sovelluksen tilanhallinnassa oli useampi kompastuskivi, jossa huolimattomuudesta johtuen ei käytetty asynkronisia kutsuja, sekä muutama kutsu, jossa palvelinpuoli ei tunnistanut haluttua polkua. Näiden korjaaminen onnistui nopeasti, sillä tilanhallinta oli tuttu konsepti, mutta huolimattomuudet ohjasivat uusien testien luomiseen. Palvelimen mallien jatkuvat muokkaukset ja pilvipalvelun käyttöoikeuksien muuttuminen uuden käyttäjätarkistuksen johdosta kuitenkin rikkoivat paikalliset testit täysin. Ongelman selvittämiseen käytettiin useampi päivä, mutta testaus jouduttiin jäädyttämään aikataulujen jäädessä jälkeen. Testaus toimi vielä hetkellisesti jatkuvan integraation putkessa, kunnes niissä muodostui uusi ongelma pöytien synkronisoitumattomuuden johdosta. Jatkuvan integraation putki ei suostunut päivittämään sovellusta, vaikka suora työntö pilvipalveluun osoitti palvelun toimivuuden, joten nekin jäädytettiin väliaikaisesti.

Ennen lomakkeiden luontia käytiin palvelimella vielä varmistamassa kutsujen toimivuus, joka aiheutti suuren hidastuksen sovelluksen kehityksessä. Palvelinpuolella ei yksittäisten noodien hakeminen onnistunut ja ongelman kanssa taisteltiin monta tuntia, kunnes paljastui kyselyn otsikossa kirjoitusvirhe. Edellä mainitun lisäksi myös poistaminen tuotti vaikeuksia johtuen tietokannan rekursiivisesta näkymästä. Näkymät eivät itsessään ole tauluja ja niistä ei voida poistaa, joten PostgreSQL halusi tietää mitä kyseiseen tauluun kohdistuneet poistot tarkoittavat. Tietokantaan luotiin uusi sääntö ”ON DELETE DO INSTEAD”, jossa ohjattiin vertaamaan jälkeläiskolumnin id numeroita noodi taulussa oleviin avaimiin ja poistamaan vain jälkeläiset. Koska aihe oli uusi ja kysely tavanomaista haastavampi, jouduttiin tietokantaan luomaan ja poistamaan useampi datapiste, jotta haluttuun toiminnallisuuteen päästiin. Tilanhallintaa ylläpitävää koodia jouduttiin myös muovaamaan niin, että kaikki poistossa karsiutuneet jälkeläiset ilmoitettaisiin myös käyttäjälle.

Toinen mahdollinen ongelmakohta nousi esille koskien sovelluksen tilanhallintaa. Tilanhallinta oli rakennettu pitämään kaikki noodit listattuna ja vaikka tämä ei ole huono idea, sen riittävyys tämän sovelluksen kannalta oli kyseenalaista. Sovelluksen työskentelyikkunassa olisi näkyvillä aktiivinen noodi ja muiden noodien tilan tietäminen oli toissijaista. Käyttäjä työskentelisi noodien kanssa yksi kerrallaan ja sen vanhemman ja lasten tulisi olla myös tiedossa, jotta ylä- ja sivupalkki pystyisivät pitämään itsensä ajankohtaisena. Ongelmaa yritettiin ratkaista antamalla kaikkia noodeja hallitsevalle tilalle myös tieto aktiivisesta noodista, tehty muokkaus onnistui, mutta tiedon päivittäminen oli työlästä ja aktiivisen noodin vanhempien etsiminen jouduttiin toteuttamaan selainpuolelta. Näiden heikkouksien takia toteutettiin tilanhallinta myös aktiiviselle noodille, joka vastaisi myös vanhempien noudosta (Liite 1, Liite 2, Liite 3, Liite 4.). Django-kehys tarjosi tähän jo valmiin kyselyn, joten hakuun riitti vain uuden polun luominen, josta tieto tarjotaan selaimelle. Palvelinpuolen hakua muotoiltiin niin, että polusta saataisiin kaikkien vanhempien tunnistusnumerot ja nimet. Palvelimen antama vastaus myös muutettiin objekteja sisältäväksi listaksi, koska se oli kaikista yhteensopivin tilanhallinnan kanssa. Selainpuolen koodia muovattiin käyttämään enemmän globaalia tilaa ja komponenttien sivustokohtaista hallintaa vähennettiin. Muutos helpotti komponenttien luontia, koska noodien tilaa ei tarvitse ketjuttaa komponenttien välillä ja koodin määrä väheni. Toisaalta jokainen komponentti noutaa nyt tilansa aggressiivisesti, pyytäen uutta tietoa usein ja globaalisti,

johtaen komponenttien hitaampaan käynnistymiseen jo pienen tietomäärän kanssa. Tämän muutoksen aiheuttama verkkaisuuden johdosta, sovelluksessa latauksia osoittavien viestien määrää kasvatettiin, jotta käyttäjäkin ymmärtäisi sovelluksen tilan paremmin.

Tämä kokonaisuus, joka selainpuolella päivitettiin, johti pilvipalvelun tarjoaman version rikkoutumiseen. Sovelluksessa käytetty jatkuvan integraation putki ei napannut virheitä, koska sovellus ei kaatunut kokonaisuudessaan, se toimi vain puutteellisesti näyttäen käyttäjälle valkoista ikkunaa, kirjautumisen jälkeen. Selainpuolella käytettävien testien määrä oli lähes olematon, ja ne eivät testanneet sovellusta käyttäjän näkökulmasta, jolloin virheitä pääsi tuotantoon. Selainpuolelle luotiin lisää testejä käyttäen Jest- ja Cypress-kehyksiä, jotka on luotu selainpuolen testaamista varten. Jest-kehys vastasi sovelluksen tilan eheyteen liittyvistä testeistä, Cypress simuloi käyttäjäkokemusta varmistaen sivun piirtymisen. Molemmat kehykset linkitettiin jatkuvan tuotannon putkeen, jotta vastaavia tilanteita ei syntyisi enää. Cypress-kehyksen käynnistys ja integroiminen tuotti suuren määrän vaikeuksia tuotantoputkessa, joten kehyksestä luovuttiin ja piirtämisen vastuu siirrettiin Jest:ille.

7.6 Moderni käyttöliittymä

Sovelluksen tilanhallinnan ja tietokannasta noudettavien kutsujen voitiin olettaa toimivan, joten sovelluksessa siirryttiin käyttöliittymän kehittämiseen. Sovelluksella oli tähän pisteeseen tultaessa kehitetty luuranko, joka mahdollisti käyttäjän kirjautumisen sisään hyvin pelkistetysti. Koska haluttiin seurata ketterän kehityksen toimintatapaa, oli seuraavan askeleena hioa etusivu ja siinä näytettävät kirjautumis- ja rekisteröitymislomake parempaan kuntoon. Sivuston esteettinen viimeistely, kuten värimaailma ja fontit, käydään läpi tilauksen tekijän kanssa, kun sovelluksen kehittäminen on lähempänä loppua ja ulkonäölliset muutokset saadaan demonstroitua reaaliajassa.

Koska sovelluksessa haettiin mahdollisimman kevyttä ja intuitiivista lähestymistapaa tiedon lisäämiseen ja poistamiseen, jouduttiin lähestymistapoja suunnittelemaan pidempään. Sovelluksessa toteutuva datan lisääminen suunniteltiin niin yksinkertaiseksi, että käyttäjä ei huomaisi edes lisäävänsä uutta datapistettä. Sovelluksen ydin on tiedon nopeassa

löytymisessä ja tämän vaatimuksen toteuttamiseksi käyttäjien on uskallettava lisätä uusia tietopisteitä. Tietopisteiden lisäämistä ei myöskään haluttu sitoa suuren lomakkeen täyttämiseen, joka voisi saada käyttäjän empimään tietopisteen lisäämistä.

Tämän lisäksi oli myös annettava mahdollisuus korjata käyttäjän tekemät virheet ja antaa mahdollisimman nopea tapa muokata näkyvää datapistettä ja sen mahdollisia jälkeläisiä, jotta käyttäjän rytmi ei rikkoutuisi liikaa mahdollisten virheiden kohdalla. Edellä mainitut ehdot kuulostavat hyvältä, mutta kehittämisen kannalta aiheuttavat paljon vaikeuksia. Monen omaan mahdottomuuteensa kaatuneen idean jälkeen syntyi idea antaa käyttäjälle kolme tärkeää painiketta, joista yksi sallisi datan lisäämisen tietopisteelle. Toinen painike avaisi käyttöön muokkaustilan, jossa käyttäjä pystyisi valitsemaan poistettavat kohteet tai päivittämään näkyviä kenttiä. Viimeinen nappi antaisi mahdollisuuden luoda uusia tietopisteitä, jotka olisivat sidoksissa johonkin listaan, jonka käyttäjä on luonut, mahdollistaen samalla sidoksen vanhemman ja lapsen välille. Samalla kun painikkeiden looginen sijainti valikoitui sovellukseen, muovattiin myös niiden ympärillä olevien elementtien vaatimaa tilaa ja ulkonäköä ja saatiin hyvin ympäripyöreä käyttöliittymä muodostettua (Kuva 20.).

Kuva 20. Sovelluksen layout ja muokkaamiseen tarkoitetut painikkeet.

Noodinäkymän yläosaan on tarkoitus luoda murupolku, josta näkyisi lähtöpisteen ja kyseisen noodin välille jääneet datapisteet, datapisteen nimeä klikkaamalla käyttäjä ohjattaisiin kyseiseen pisteeseen. Vasemmalle puolelle, sivun yläkulmaan tulee sovelluksen logo ja sen alle muodostuisi lista kaikista näkyvän tietopisteen jälkeläisistä. Kuten yläpalkin nimiä, myös kyseisen sivupalkin datapisteitä voidaan käyttää navigointiin. Sivun keskelle muodostuu

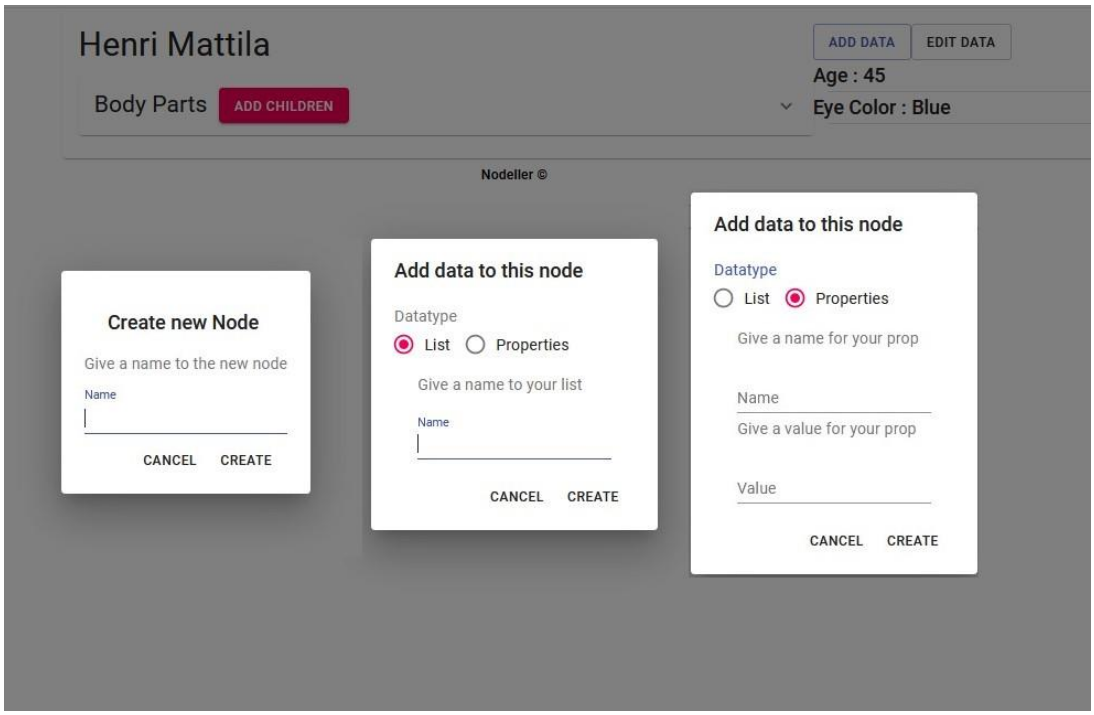
aktiivisen noodin sisältämä tieto. Jokainen noodi, sisältäisi oman datansa ja kyseinen data on tarkoitus näyttää omassa näkymässään. Noodin datassa ilmentyvät listat ovat kehyksen vasemmalla puolella ja yksittäiset avain-arvo-parit löytyvät oikealta. Painamalla Listan nimessä vieressä olevaa Add Children -painiketta, voidaan näkyvälle noodille luoda lapsi. Sivulle muodostetaan hyvin yksinkertainen dialogi, joka sitoo käyttäjän vain nimeämään muodostettavan noodin ja poistuu sitten näkyvistä. Tiedot välitetään palvelimelle, ja vastauksen saatuaan sivu piirtyy uusiksi, näyttäen uuden datapisteen nimen, kun listan säiliö on laajentuneessa tilassa. Käyttäjä pidetään aktiivisen noodin ikkunassa, eikä häntä pakoteta liikkumaan juuri kehitetyn noodin näkymään, jolloin aktiiviseen noodiin saadaan huomattavasti nopeammin uutta sisältöä. Olisi ollut huomattavasti helpompaa pakottaa käyttäjää lisäämään kaikki tulevassa noodissa sisältyvä tieto samassa ikkunassa ja sitoa noudattamaan omia päätöksiä, tämä ratkaisu olisi kuitenkin sotinut sovelluksen suunnittelun ehtoja vastaan.

7.6.1 Interaktio ja lomakkeet

Edit Data -painike antaa käyttäjälle mahdollisuuden muokata noodissa näkyvää dataa. Painaessa kyseistä painiketta, ikkunassa tehdään vain pieniä muutoksia, joka luo efektin siitä, että sivu vain päivittää itsestään pieniä osia. Tätä tekniikkaa hyödynnetään monissa moderneissa sovelluksissa se luo illuusion natiivista sovelluksesta ja peittää menneisyydestä tutut sivustojen latautumiset. Muokkausnäkymässä käyttäjälle annettiin mahdollisuus valita listanäkymän noodeja poistettavaksi ja päivittää tietopareja kirjoittamalla avaimille ja arvoille uudet sisällöt. Kun käyttäjä kokee olevansa tyytyväinen muutoksiin, sivusto päivitetään vasemmasta alakulmasta löytyvällä painikkeella. Tämän idean toteuttaminen ei todellakaan ole yksinkertaista ja tiedon on liikuttava saumattomasti useamman komponentin välillä, kyseisen näkymän ongelmia avataan seuraavassa kappaleessa enemmän. Viimeinen ja loogisesta näkökulmasta katsottuna tärkein palanen on uuden tiedon lisääminen näkyvään noodiin. Tämä painike mahdollistaa kyseisen näkymän luomisen, ja onkin ainoa rajoitus, mikä käyttäjälle annetaan datan lisäämisen suhteen. Tämä rajoittava tieto on lisättävän tiedon arvojen määrä. Käyttäjän valitessa listan, häneltä pyydetään nimeä listalle ja vahvistettu valinta lisää aktiivisen näkymän listaosioon uuden lohkon. Jos halutaan vain yksi arvo, käyttäjän annetaan määrittää avain ja sille vastaava arvo. Vahvistuksen

jälkeen tieto lähetetään palvelimelle ja pari piirretään oikealla näkyvien parien jatkoksi. Mikään edellä mainituista painikkeista ei siirrä näkymää pois aktiivisesta noodista, tieto lähetetään palvelimelle ja vahvistuksen jälkeen selain piirtää näkymän päivitetyllä datalla. Kehittäessä sovellusta paikallisesti, nämä operaatiot vievät toivottua enemmän aikaa, joten sovelluksessa näkyviä ilmoituksia lisättiin näiden tapahtumien ympärille. Pilvipalvelimella sijaitsevassa tuotantoversiossa latausajat ovat huomattavasti lyhyempi, syyksi voidaan olettaa palvelimen ja palvelimessa löytyvän tietokannan käyttävän välimuistia tai kommunikoivan vapaammin kuin paikallisesti kehittäessä. Painikkeiden lopullinen toiminnallisuus on käyttäjälle ystävällinen ja antaa nopean kehittämisen tunteen, vaikka taustalla tapahtuva logiikka kasvoi paikoitellen hyvin hankalaksi ymmärtää. Kun sovellukseen ruvetaan kehittämään omaa teemaa, tullaan muokkauksen ja selailun välille saavuttamaan selkeä ero, ja käyttäjä pidetään paremmin tietoisena sovelluksen tilasta rikkomatta modernia ulkonäköä (Kuva 21.).

Kuva 21. Datan lisäämisessä näytettävät dialogit.



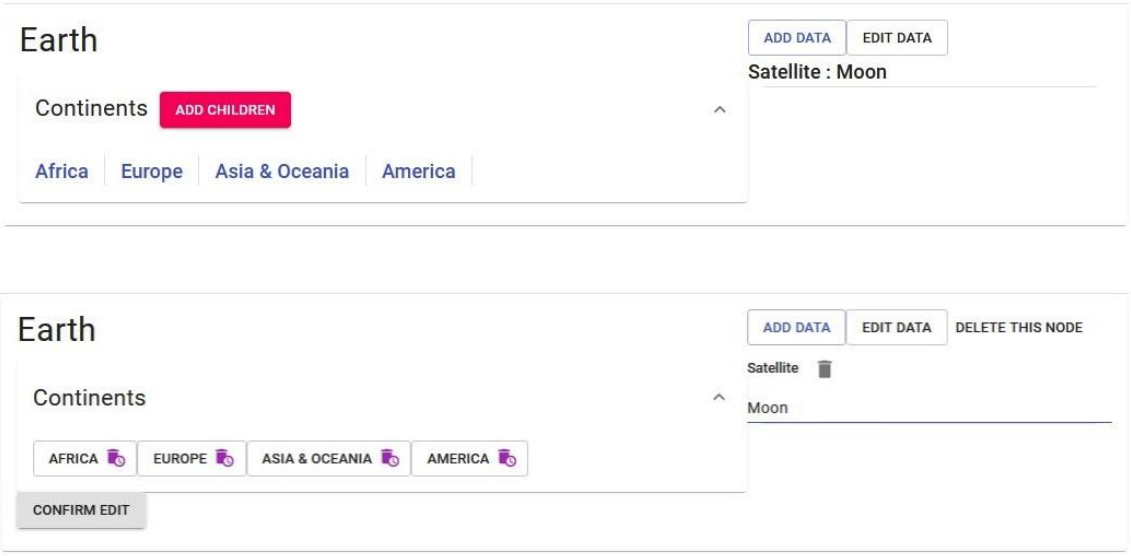
Moni pitää React-kirjaston heikoimpana lenkkinä lomakkeiden luomista ja ohjelmaa luotaessa tämä heikkous hidasti monesti tahtia ohjelman edistyessä. Yksinkertaiset lomakkeet toteutuvat suhteellisen vaivattomasti, jos ohjelmoijalla on aikaisempaa

kokemusta web-sovellus kehityksestä. Laajemmat lomakkeet lisäävät koodin määrää merkittävästi ja jokainen lomake luodaan kopiona entisestä, mutta lomakekentän yksittäiset parametrit pitää muovata vastamaan haluttua syöttöä. Lomakkeiden tilojen hallitsemisessa voidaan nykyään käyttää paikallista tilaa ja kyseinen päivitys helpotti lomakkeiden suunnittelua merkittävästi, sillä jokainen komento voidaan tallentaa paikalliseen tilaan resurssitehokkaasti ja hyvin pienellä määrällä ohjelmointia.

Tätä sovellusta kehittäessä nousi esille useampi kohta, jossa lomakkeen tekeminen ja sille annettujen komentojen syöttäminen paikalliseen tilaan olisi ollut huomattavasti helpompaa, jos vaaditut kentät olisivat ennalta tiedossa. Edellä mainittu ongelma korostui muokkausnäkymässä tarkoitetun lomakkeen luonnissa. Jokaiselle kentälle annettiin järjestyksessä yksi avain ja sen arvo sekä lista kaikista pareista, joita käyttäjä oli kyseiselle noodille luonut. Teoriassa kenttien piti näyttää yksi pari, ja tämän jälkeen kuunnellaan käyttäjän komentoja. Käyttäjän päivittäessä kenttää, kenttä etsisi alkuperäisestä listasta vanhan datan, ylikirjoittaisi sen ja lähettäisi päivitetyn versionsa noodinäkymä-komponentille. Alkuperäisessä toteutuksessa annettiin mahdollisuus muokata sekä avainta että arvoa mielivaltaisesti. Tämä olisi ollut toteutettavissa mutta vanhan tiedon korvaaminen olisi vaatinut huomattavan määrän analysointia ja lopullisessa versiossa pitäydyttiin vain mahdollisuudessa muokata arvoa. Yksittäisiä parametrejä piti myös pystyä poistamaan ja avainten viereen luotiin pieni ikoni, jota klikatessa kyseinen arvo siirrettiin poistettavien listalle ja muokkaaminen estettiin. Listalla olevien jälkeläisten hallinta oli helpompi toteuttaa, koska muokkaus tarvitsi vain tiedon identifiointinumerosta, joka välitettiin takaisin kutsuvalle komponentille, joka puolestaan poisti alkuperäisestä listasta valitut numerot ja lähetti päivitetyn tiedon palvelimelle. Jos sovelluksessa olisi päädytty yksittäisten tietojen järjestelmälliseen poistamiseen, olisi logiikan toteuttaminen ollut paljon suoraviivaisempaa. Käyttäjä olisi valinnut poistettavan kohteen, tämä tieto olisi välitetty palvelimelle ja näkymä olisi päivittynyt uusimalla versiolla noodista. Joiltain ongelmilta olisi vältytty, jos olisi joustettu kehyksen standardeista. React-kehys perustuu komponenttien luomiseen, jossa logiikka nivoutuu mahdollisimman tiukasti yhden komponentin sisään ja tämän säännön noudattaminen aiheutti suuren määrän datan siirtoa komponenttien välillä, miltä olisi vältytty niputtamalla useampi komponentti yhteen muokkaustilaa toteuttaessa.

Logiikan valmistuessa ja päivitetyn ohjelman löytyessä pilvipalvelimesta sain työnohjaajalta kiitosta ripeästä toteutuksesta, samalla kuitenkin sain palautetta käyttäjälle näkyvissä olevan noodin poiston mahdottomuudesta. Tehty toiminnallisuus salli vain näkyvillä olevan noodin jälkeläisten poiston, eikä näkyvissä olevaa noodia pystynyt poistamaan. Tälle ominaisuudelle luotiin kentän oikeaan reunaan painike ja kyseinen painike paljastui vasta muokkaustilassa (Kuva 22).

Kuva 22. Ylempänä tavallinen näkymä ja sen alapuolella muokkausnäkymä.



Toiminnallisuus vaikutti helpolta lisäykseltä, mutta lisäystä valmistaessa, huomattiin yllättäviä vaikutuksia, jotka vaikeuttivat komponentin suunnittelua. Käyttäjän poistaessa näkyvän tietopisteen oli myös reagoitava sen vanhemman tietoihin ja käyttävä vanhemman lista läpi, poistaen mahdolliset viittaukset tähän datapisteeseen. Ennen poistoa käyttäjälle myös piirretään kuva poistetun tietopisteen vanhemmasta, jotta poiston jälkeen käyttäjällä olisi selkeä piste, josta jatkaa työskentelyä. Lopullinen kutsu haki ensin vanhemman, jonka datasta poistettiin viittaukset poistettuun jälkeläiseen. Datan muokkauksen valmistuttua, näkyviin piirtyi päivitetty vanhempi ja taustalla poistettiin aikaisemmin kuvalla näkynyt lapsi.

Toteutus onnistui kiitettävästi, ja poisto, sekä uuden näkymän piirtyminen näyttää luonnolliselta. Tämän muutokset vaikutukset toteutuivat kuitenkin välittömästi, eikä käyttäjä

saanut miettimisaikaa, joten luotiin uusi dialogi, joka vahvistaa vielä käyttäjältä hänen poistamisaikeensa ja varoittaa poistamisketjusta, jos noodilta löytyy jälkeläisiä. Vahvistusdialogi luotiin niin, että molemmat poistot hyödyntävät sitä ja kirjoittavat poistoja vastaavat dialogit, noudattaen kehyksen hyviä toimintatapoja. Pääikkunaa piirtäessä kehys hakee näkyvillä olevan noodin raakadatan, analysoi sen ja luo tarvittavan näkymän. Tämä tarve toistui myös vanhemman kohdalla, mutta olemassa olevaa koodia ei suoraan pystytty käyttämään uudelleen. Jouduttiin paheksuttuun tilanteeseen, jossa koodia toistetaan useammassa kohtaa ja tätä on pyrittävä välttämään. Ongelman korjaamiseksi jouduttiin luomaan funktio, joka toimii geneerisemmin ja palauttaa puhdistetun datan välittämättä sen tyypistä ja riippumattomana mistä sitä kutsutaan. Tämä korjaus puhdisti myös noodinäkymää ja helpottaa uusien ominaisuuksien luomista tulevaisuudessa.

7.6.2 Tyyli ja teema

Jotta sovelluksen tyyli saadaan pidettyä yhtenäisenä, on ohjelmoijan hyvä määritellä standardit, joita sivu seuraa. Moderneissa kehitystyökaluissa on yleensä teeman asettamiselle oma resurssilohkonsa, jossa määritellään vakiovärit, fontit ja mahdolliset muut sivustolla yleisesti esiintyvät elementit. Material UI -kirjasto antaa mahdollisuuden luoda oman tyylitiedoston, johon määritellään elementtien oletustietoja, joita on helppo liittää komponentteihin ohjelmaa kehittäessä. Asettamalla elementteihin samankaltaisuuksia saadaan sivustolle selkeä linjaus, ja ohjelmassa tapahtuvat dynaamiset sivustopäivitykset näyttävät luonnollisilta. Kirjastoa ylläpitävällä kommuunilla on loistavia työkaluja, jotka helpottavat teeman luomista ja tässä sovelluksessa käytettiin Material UI theme creator -nimistä työkaluja, joka renderöi käyttäjän tekemät muutokset suoraan työkalun omalla sivustolla ja tarjoaa täytteen tyylitiedostolle. Sovelluksen lopullinen visuaalinen näkemys ja väritys sovittiin tarkentuvan toimeksiantajan kanssa mahdollisuuksien mukaan. Alustavaksi väritykseksi sovittiin tumma teema, jossa olisi muutama huomioväri kriittisissä pisteissä.

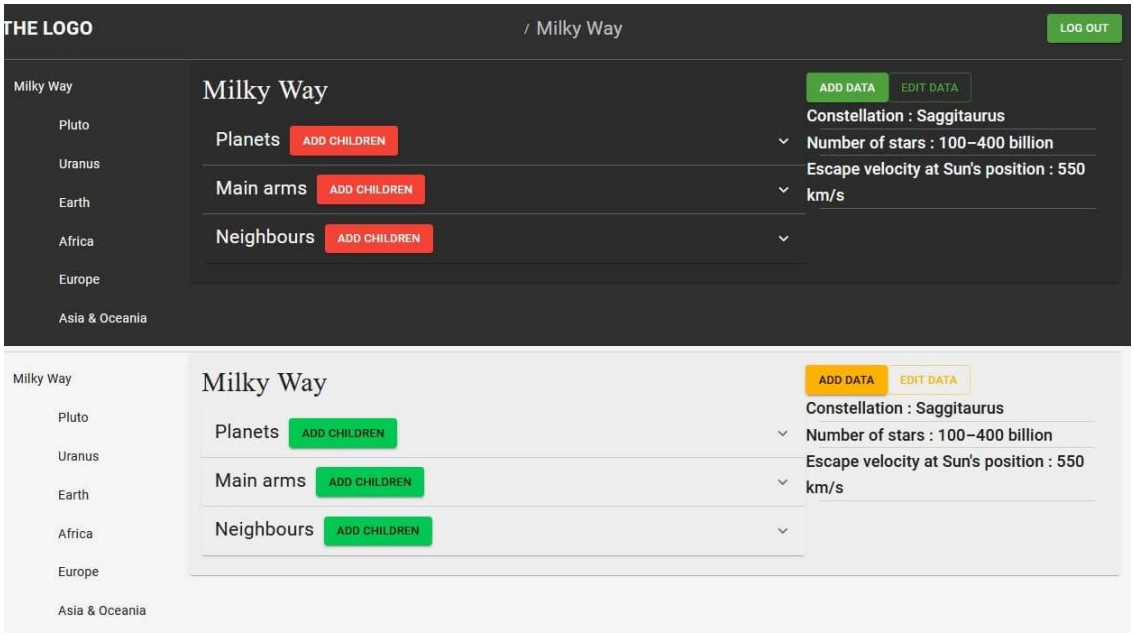
Teeman asettaminen on suoraviivaista, vaikka ongelmia usein ilmeneekin tiettyjen komponenttien kohdalla. Teemasta luodaan oma objekti, jolle annetaan oletusarvoisia parametrejä. Material-UI -materiaalit osaavat oletusarvoisesti noutaa parametrien takana löytyviä värejä ja teemassa määritetyt vaihtoehtoiset värit saadaan käyttöön yksinkertaisilla

tekstikomennoilla. Teemalle luodaan tarjoilija ja tämä tarjoilija käärii projektin juuren, jolloin se on käytössä kaikilla komponenteilla. Tätä työtä tehdessä teema ei tarttunut kuin muutamaan komponenttiin, ja pitkään kestäneen tutkimisen jälkeen ongelma saatiin paikallistettua versiointiin. Tätä työtä tehdessä on tyylistä vastaavasta Material-Ui:sta julkaistu versio viisi. Kyseinen versio aiheutti todella suuria muutoksia vanhaan tapaan toteuttaa tyylittely, ja kirjaston ylläpitäjät ovat julkaisseet ison dokumentoinnin muutoksista, joita käyttäjät joutuvat tekemään päivittääkseen sivustonsa. Sovelluksessa käytetyt komennot, joilla materiaaleja noudettiin, olivat sekoitus uutta ja vanhaa kutsua. Sovelluksen tekoon käytetyssä IDE:ssä oli ladattu kirjaston käyttöä nopeuttava aputyökalu, joka täytti kirjastokutsut vanhalla menetelmällä, ja nämä kaikki kutsut oli päivitettävä. Kutsujen päivitys ei kuitenkaan päivittänyt kaikkien komponenttien väritystä, kuten sivuston valkoinen taustaväri. Kirjaston dokumentaation teeman käsittelyyn liittyvästä osiosta ei löytynyt ongelmalle vastausta, joka yllätti monelta osin, sillä tämä on hyvin olennainen osa websovelluskehitystä. Lopullinen syyllinen on CSS Baseline-nimisen komponentin puuttuminen, jonka tarkoituksena on antaa yhtenäinen linja sovellukselle. Ongelman takia käytiin läpi monta videota ja ohjetta aiheeseen liittyen, mutta missään näistä ei asetettu sivustolle tätä komponenttia, joten ongelma saattoi olla omalla kohdalla poikkeuksellinen, joskin syy aiheutuneelle jäi epäselväksi. Kyseisen komponentin asettaminen teeman ja ruudulle piirtyvän komponentin väliin kuitenkin ratkaisi ongelman ja kaikki teemassa asetetut värit saatiin näkyviin. Osa päivitetyistä komponenteista oli kokenut myös parametrillisiä muutoksia, jotka heijastuivat sivustolla esimerkiksi listojen suuntauksen vaihtumisena. Hajonneet parametrit kirjoitettiin uusiksi ja toiminnallisuus saatiin palautettua onnistuneesti.

Päivitetty ulkonäkö esitettiin aiheen keksijälle Markolle, joka oli tyytyväinen muutoksiin, mutta pyysi mahdollisuutta antaa käyttäjän vaikuttaa teemaan. Moderneissa sovelluksissa on usein mahdollisuus valita valoisan ja tumman teeman välillä ja tämä mahdollisuus päätettiin antaa myös tässä työssä. Ajatuksena oli luoda näppäin, jolla käyttäjä voisi vaihtaa teeman tyyliä saumattomasti lataamatta sivustoa uudestaan. Tyylikirjasto on tukenut tätä muutosta jo pidempään, mutta ominaisuus on ollut raskas sivustolle, mikä on näkynyt suorituskyvyn laskuna. React versiossa 16.8 julkaistujen koukkujen mukana on tullut UseMemo-niminen koukku, joka mahdollistaa kyseisen ominaisuuden luomisen laskematta

suorituskykyä. Koukku tallentaa sivuston hetkelliseen muistiin ennen uudelleenpiirtämistä ja päivittää vain ne osat, jotka sitä vaativat. Tämä ominaisuus on ideaalinen tehtävään, jossa haluamme pitää näkyvät komponentit ja muuttaa vain niiden väritystä. Teema objektin tila riippuvuus, jonka muutos päivittäisi objektin väriin liittyvät parametrit, teksti- ja komponenttiparametrit, jäivät tilan ulkopuolelle ja säästivät lisää suorituskykyä. Sovelluksen juureen luotiin painike, jota painamalla annetaan kutsu komponentille viedä päivitetty tila teemaobjektille, joka puolestaan analysoi sen ja palauttaa halutun värityksen sovelluksen juureen (Kuva 23.).

Kuva 23. Sovelluksen tumman ja vaalean teeman erot.



Ominaisuutta testattiin painetestillä, jossa painiketta painettiin monta kertaa lyhyen ajan sisällä. Muistikoukku toimi hyvin ja sivuston todettiin pitävän suorituskykynsä, vaihdon ollen jopa odotettua saumattomampaa. Tämä tapahtuma kuitenkin paljasti odottamattoman lataamisen toteutuvan hyvin usein sivuston kirjautumisessa vastaavassa näkymässä. Teeman vaihtaminen laukaisi tapahtuman, jossa käyttäjä noudetaan sivuston paikallisesta muistista, jota muistia säästävä koukku ei analysoinut ja tämä toteutui jokaisella piirtymiskerralla. Tämä ei luonnollisesti ollut haluttu ominaisuus ja kirjautumisesta vastaava näkymä korjattiin tekemään useampi tarkistus ennen paikallisen muistin tutkimista. Muutos oli merkittävä ja sen vaikutus näkyi myös päänäkymässä, jonka ei tarvinnut odottaa juurikomponenttien

tarkistuksia (Liite 5, Liite 6, Liite 7.). Käyttöliittymän testaukset pakottivat myös pieniin värimuutoksiin ja esimerkiksi vaaleassa teemassa huomiovärinä toiminut keltainen koki useamman muutoksen johtuen sen huonosta näkyvyydestä taustaa vasten.

8 Viimeistely

Sovelluskehityksessä oli päästy tilanteeseen, jossa logiikka toimi oikein selaimessa ja palvelinpuolella. Palvelimen ja selaimen välinen yhteistyö oli saumatonta, selaimen lähettämät pyynnöt ja päivitykset löysivät paikkansa tietokannasta palvelinpuolen avustuksella ja vastaukset piirtyivät käyttäjälle toivotusti. Sivustolla ei ollut näkyviä ongelmia ja palvelimen yhteyteen liittyvät virheilmoitukset ja ilmoitukset heijastettiin käyttäjälle. Sovelluksen ulkonäkö oli yhtenäinen ja pienet nyanssit väreissä ohjasivat käyttäjän huomion haluttuun paikkaan. Käyttäjän antamat käskyt ja niiden seuraukset piirtyivät huomattomasti ruutuun ja haluttuun modernisuus oli saavutettu. Sovelluksesta kuitenkin puuttui viimeinen silaus ja pieniä muutoksia jouduttiin toteuttamaan useammassa paikassa. Päivitysten perusteet olivat myös hyvin vaihtelevia, mikä on seurausta sovelluksen ympäripyöreästä kehittämisestä, jossa kaikkia osa-alueita rakennettiin ketterästi.

Sivustolla näkyvät komponentit vaativat kuitenkin pientä hiomista, sillä moni komponentti käytti vielä oletusasetuksia. Muutokset olivat hyvin pitkälti makuasioita, joista kävimme pitkän keskustelun Markon kanssa. Lopullisessa näkymässä melkein jokainen komponentti koki pieniä muutoksia, jotka kuitenkin sopivat sovelluksen identiteettiin, ja antoivat käyttäjälle vielä selkeämmän kuvan tiedon muodostamasta rakenteesta. Yläpalkissa noodien siirtymistä kuvaavat ikonit päivittyivät nuoliksi, jotka toivat paremmin esille hierarkian suunnan. Sivuston vasemmalla laidalla näkyvää jälkeläisten listaa hiottiin korostuksiensa puolesta ja lopputulos on puhtaampi. Suurimman uudistuksen koki noodin henkilökohtaisista parametreistä vastaava komponentti, joka ei täyttänyt alkuperäistä ideaansa täysin. Omat parametrit käärittiin korttikomponentin sisään ja teksti jaoteltiin selkeään otsikkoon ja alatekstiin. Muutos johti selkeämpään ulkoasuun, jossa tekstikenttä pystyi kasvamaan suuremmaksi rikkomatta sovelluksen dynamiikka ja ulkoasua. Samalla arvolle asetettiin maksimikorkeus, jonka jälkeen sisältö muuttuisi vieritettäväksi ja pitäisi sivun pituuden kurissa.

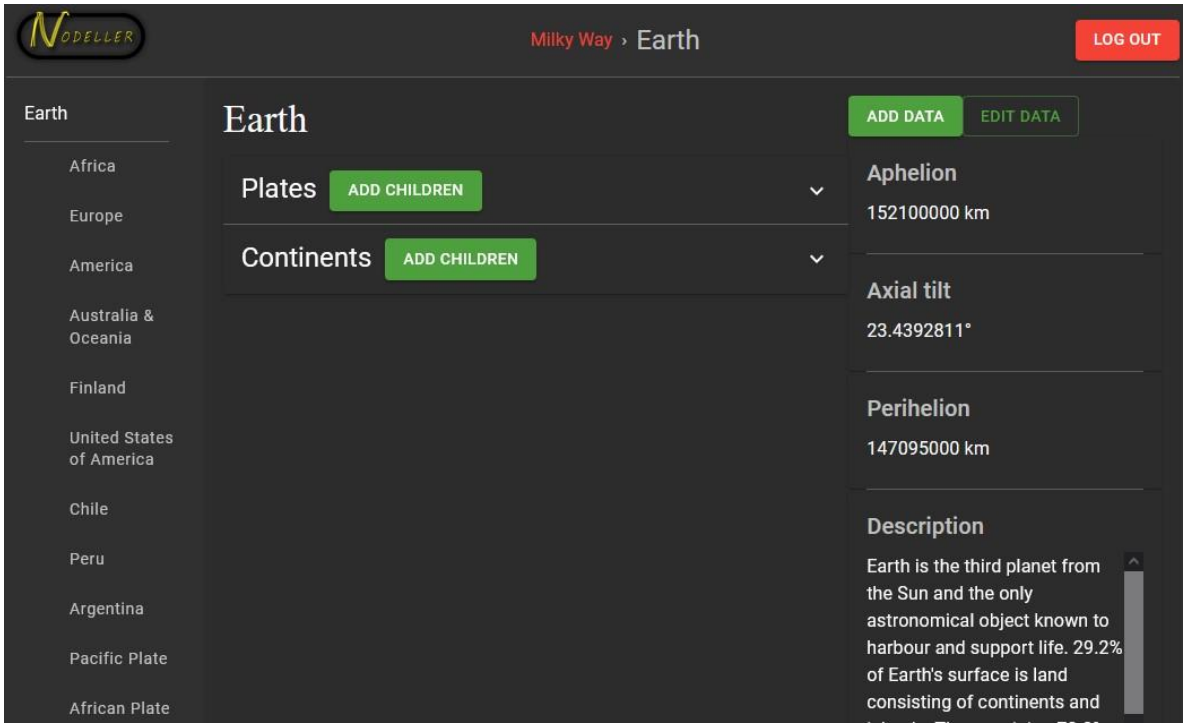
Testaamisen aikana myös lomakkeissa huomattiin puutteita. Lomakkeet eivät sisältäneet ollenkaan tarkistuksia ja käyttäjä oli mahdollista luoda nimettömiä avaimia sovellukseen, jotka puolestaan estivät kyseisten tietojen muokkaamisen. Kirjastoa käytettäessä päädytään usein käyttämään apukirjastojen lomakkeiden tarkistusta ja varsinkin suuremmissa lomakkeissa sitä pidetään oikeana tapana hoitaa validointi. Tässä projektissa lomakkeet olivat kuitenkin yksinkertaisia ja haluttiin kokeilla kirjaston omaa virheiden käsittelyä. Apufunktioihin luotiin tarkastava funktio, joka käsitteli käyttäjän antaman syötteen, ajoi sen sääntökoneiston läpi ja palautti totuusarvon sovelluksen käyttöön. Kyseisetä funktiosta saatiin niin geneerinen, että sitä pystyttiin käyttämään kaikissa lomakkeissa, mikä puolestaan on kirjaston käytön hyvien tapojen mukaista. Lomakekompotteihin lisättiin uusi tila, jonka tehtävänä oli kartoittaa funktiolta saatu vastaus ja virheen sattuessa näyttää käyttäjälle virheviesti ja maalata kenttä punaisella osoittaakseen virheet selkeästi käyttäjälle.

Sovellusta suunniteltaessa käyttäjien odotettiin kommunikoivan selaimen kanssa pääasiassa tietokoneella. Puhelimien kehittyessä yhä useammat sivut kuitenkin kehitetään myös mobiililaitteille sopivaksi ja mobiilivierailuiden määrä sivustoilla on jopa ohittanut tietokoneella suoritettavat vierailut (Enge, 2021). Sovelluksen reagoiminen käyttäjän laitteeseen oli jäänyt pienelle huomiolle kehityksen aikana ja tämän korjaamiseen käytettiin merkittävä määrä aikaa projektia viimeisteltäessä. Jokainen näkymä käytiin yksitellen läpi vaihtaen käyttäjän näytön kokoa. Kehittäessä ohjelmaa pystyimme kokeilemaan sovellusta vain muutamalla oikealla puhelimella, mutta useammat modernit selaimet antavat mahdollisuuden katsoa sivuja muiden laitteiden näkökulmasta, helpottaen kehitystyötä. Nykyisin on niin monta laitetta, jolla sivuja voidaan selata, että korjaamiseksi on kehitetty useampia eri työkaluja, jotka voivat nopeuttaa etenemistä. Teoriassa jokaisesta komponentista ja näkymästä pitäisi luoda variaatiot kaikille mahdollisille näytöille, jos haluttaisiin saavuutta täysi laitteistotuki. Täydellisen tuen toteuttaminen on kuitenkin liian pitkä prosessi ja moni ohjelmoija tyytyy asettamaan tietyin väliajoin toteutuvia piirrettävän kuvan muokkaavia mediakyselyitä. Näkymiä luodessa asetteluun käytettiin paljon Grid-nimisiä komponentteja, syy tähän on niiden hyvä muovautuvuus tilanteesta riippuen. Komponentit sallivat rikkoutumispisteiden määrittelyn ja pystyvät näin muovautumaan

käytetyn ikkunan mukaan. MaterialUi-kirjastolle on oletuksena asetettu vakiopisteet tietyin pikselivälein, nämä ovat kuitenkin täysin vaihdettavissa ja halutettaessa ne voidaan määritellä esimerkiksi teemaobjektiin. Esimerkiksi ikkunan koon alittaessa vakiona löytyvän sm-koon arvon 600 pikseliä, voidaan komponentille määrätä käytettävän ikkunan täysi leveys. Jos ikkunaa sen sijaan kasvatetaan yli asetetun rajan, voidaan komponentin leveys puolittaa ja mahduttaa viereen toinen komponentti. Kirjaston tarjoamat komponentit ovat jo itsessään hyvin reagoivia, vaikka mobiilivalmiuteen ei keskityttäisiinkään, harvoin ylettäen kuitenkaan hyvälle tasolle ilman tarkennettua optimointia. Suurimmaksi ongelmaksi muodostui sivupalkin vaatima leveys, joka murtautui noodikomponentin päälle tilanpuutteesta johtuen. Sivupalkille asetettiin ehto siirtyä noodinäkymän päälle ja jälkeläisiä sisältävälle listalle asetettiin maksimipituus ja vieritysmahdollisuus. Murtumispisteeksi asetettiin 600 pikseliä, joka on hieman keskiverto tablettia pienempi ikkuna, tarkoituksena pakottaa lista päänäkymän päälle mobiililaitteilla.

Sovelluksen kriittisimmässä näkymässä, joka sisälsi näkyvän noodin tiedot, pakotettiin omat parametrit komponentti ja listakomponentti siirtymään toistensa päälle jo tablettien kokoisissa ikkunoissa. Accordion-komponentti syyllistyi myös liialliseen horisontaalisen laajentumiseen ja se kiedottiin Grid-komponentin sisään, mikä mahdollisti useamman jälkeläisten jatkamisen seuraavalle riville. Samassa komponentissa ilmeni myös painikkeiden laajentumista useammalle riville mobiilinäkymässä. Korjaustoimenpiteeksi asetettiin rikkoutumisraja mobiilinäkymään, joka aiheuttaisi tekstipainikkeiden muuttumisen pelkiksi ikoneiksi. Kyseinen komponentti jäi kuitenkin hieman alttiiksi painikkeiden laajentumiselle tilanteissa, joissa listan nimi oli tavanomaista pidempi (Kuva 24.).

Kuva 24. Sovelluksen ulkonäkö projektin lähestyessä loppua.



Sovellukselta ja sen visuaalisesta ilmeestä puuttui myös logo. Otin Markoon yhteyttä ja kävimme läpi mahdollisuuksia viestintäsovellusta hyödyntäen, jossa kumpikin piirsi ideoitaan. Modernit logot ovat yksinkertaistettuja ja monesti niissä näkyy pelkkää tekstiä, tämä oli myös meidän ajatuksenamme, mutta halusimme kuitenkin pienen erotuksen normeista. Pääsimme yhteisymmärrykseen nopeasti ja ensimmäinen vedos logosta syntyi nopeasti. Ensimmäinen vedos toteutettiin Paint-nimisellä ohjelmalla, mikä mahdollisti nopean vision luomisen, joskin lopputulos ei ollut silmiä hivelevä. Vedos oli mielestämme moderni, mutta sisälsi haluamamme poikkeavuuden sanaleikillä, sovelluksemme nimi, kun oli hyvin samankaltainen englanninkielisen Noodels-sanan kanssa. Vedosta seuraten loin lisää versioita logosta vektorigrafiikkaan perustuvalla Adobe Illustrator -ohjelmalla. Tätä sovellusta käyttämällä luodut kuvat muodostuvat vektoreista ja sallivat logolle yksinkertaistetumman muokkauksen, kun vektorit muuttavat pituutta dynaamisesti. Lopullinen logo oli mielestämme onnistunut ja pitäytyi alkuperäisen vedoksen yksinkertaistetussa ideassa, luoden siihen useamman kerroksen syvyyttä. Tulimme Markon kanssa kuitenkin siihen tulokseen, että tulevaisuudessa jätämme taiteellisen luomisen osaavien ihmisten käsiin. Logo tallennettiin Firebase-nimiselle alustalle, josta se tarjoiltaisiin

käyttäjille etusivua ladattaessa. Lataus kuitenkin kesti kauan ja lukuisista optimointiyrityksistä huolimatta kuva päätettiin syöttää staattisesti. Heroku:lla on kuitenkin tapana poistaa staattista sisältöä ilmaisten sivujen kohdalla, joten Firebase-tuki jätettiin taustalle pyörimään hätävaraksi.

Sovelluksen tarkoitus oli sallia pitkienkin tarinoiden kirjaaminen sovelluksen yksittäisiin datapisteisiin ja noodien nykyinen sijainti sovelluksen oikealla laidalla ei ollut ajatuksen kanssa täysin harmoninen. Omille parametreille oli yksinkertaisesti liian vähän tilaa horisontaalisesti, jolloin teksti vaati itsellensä tilaa vertikaalisessa suunnassa. Tämä aiheutti tilanteista, joissa parametrit jatkuivat pitkälle sivustoon ja listat loppuivat niitä huomattavasti aikaisemmin. Yksittäiselle parametrille annettiin maksimipituus, jonka ylittäessä komponentti tarjoaisi vieritettävän sivupalkin, mutta tämäkään ei täysin ratkaissut ongelmaa. Marko toivoi parametri- ja listakomponenttien yhdistämistä samalle osiolle, joka osoittautuikin visuaalisesti miellyttäväksi ja toimivaksi ratkaisuksi.

9 Yhteenveto ja pohdinta

Jo sovellusta suunnitellessa kävi ilmeisen selväksi, että omat taidot on ajettava rajoille valmiin lopputuloksen saavuttamiseksi. Vaikka moni käytetyistä teknologioista oli ennestään tuttuja, päädyttiin myös rohkeasti käyttämään uusia toteutustapoja ja menetelmiä, niin kuin tällä alalla täytyykin, jos haluaa pysyä sovelluskehittämisen aallonharjalla. Projektin onnistunut lopputulos on monelta osin hyvän suunnittelun ansiota ja kokonaisuuksien pilkkomista helpommin hallittaviksi osiksi. Lähtökohtana oli luoda käyttäjälle sovellus, jota ei sitoisi mikään selkeä sääntö tai ennalta valittu päämäärä. Käyttöön haluttiin luoda sovellus, joka kieltämisen sijaan antaa vauhtia ja työntää uusien pisteiden luomiseen. Saadessani sovellusidean Markolta, ensimmäisenä ajatuksena oli sen tuomat mahdolliset haasteet, jotka olivat selvästi omaa osaamistasoa korkeammat. Lopulta halu menestyä alalla ja oman itsensä haastaminen osoittautuivat vahvemmiksi, ja huomasin miettiväni mahdollisia ongelmia ja niiden ratkaisuja vapaa-ajalla ja tartuin haasteeseen. Tästä olen erittäin kiitollinen Markolle, sillä hän sai minut antamaan parasta itsestäni tähän projektiin, ja olen varmasti parempi ohjelmoija nyt, kuin ennen kyseistä projektia. Luulin tunteneeni osan projektissa käytetyistä tekniikoista, mutta kuten useamman koulussa opetetun tekniikan kanssa, kosketus

menetelmiin ja tekniikoihin jää kevyeksi ja päästääkseen syvempään osaamiseen on niitä harjoiteltava omatoimisesti.

Monesti projektia tehdessäni tuli tilanteita, jossa halu ylittää aita matalimmasta päädystä oli suuri. Halusin kuitenkin saavuttaa sovelluksen, jota kehtaa esitellä omassa portfoliossa, joka heijastaisi omaa motivaatiota tälle alalle ja pääsin tavoitteeseen paremmin kuin osasin ennalta odottaa. Projektissa tuli useampaan otteeseen vastaan tilanne, jossa suunniteltu edistymistapa ei vain osoittautunut toimivaksi ja vaihtoehtoisia lähestymistapoja jouduttiin kehittämään projektin edistämiseksi. Henkilökohtaisesti koin, että valitsemani tekniikat sovelluksen toteuttamiseen olivat pääosin oikeita, muutamaa poikkeusta lukuun ottamatta. Näistä suurimpana oli relaatiokannan käyttö, jotka ovat kehittyneet huomattavasti lähiaikoina, mutta etsiessäni vastauksia niihin liittyviin ongelmiin löysin useamman paremman tavan toteuttaa hierarkiaa, jos tietokannaksi olisi valittu graafipohjainen kanta. Puumallin saavuttamiseksi luodut SQL-näkymät ovat toki valideja, mutta en usko niiden olevan paras tapa toteuttaa kyseinen datarakenne. Alussa suunniteltuun geneerisyyteen päästiin yli odotusten, mutta tämän saavuttamiseksi kehitetty datakolumni, johon saatiin tallennettua vaihtelevaa tietoa JSON-muodossa, ei mielestäni vastaa relaatiokannan syvintä olemusta. Django-kehys antoi myös paljon anteeksi omalla ydinlogiikallaan, joka mahdollisti jälkeläisten ja vanhempien noutamisen yksinkertaisilla komennoilla. Djangon käyttäminen Rest-kehyksenä oli uusi kokemus, mutta dokumentaatio oli erittäin toimiva ja eteen tulleisiin ongelmiin löytyi monia toteuttamistapoja ja ohjenuoria. Näen itseni luottamassa kyseiseen kehykseen ja sen Rest-rajapintaan myös tulevaisuudessa. En odottanut kehittyväni SQL-komennoissa kyseisen projektin aikana, mutta oman näkymän lisäksi pääsin tutustumaan myös kytkimiin sekä poistamisen toteuttamiseen luoduissa näkymissä, jotka molemmat mielletään ammattilaisten keskuudessa haastaviksi.

Selaimenpuolen React-kirjasto, vaikka tuttu entuudestaan, näytti omaa vahvuuttansa dynaamisten sivujen työkaluna. Kirjastoa käyttäessä sain palautettua mieleen jo opitut asiat, sekä paljon uusia koukkuja ja tapoja pitää käyttäjä kiinnittyneenä selaimeensa. Kirjaston liittäminen Django-kehykseen tuotti käyttöönotossa paljon hankaluuksia. Djangon asetuksien muuttamien ei tuottanut paljon virheviestejä, ja polkujen onnistunut muodostaminen sekä ylläpitäminen kirjaston kanssa vei kehittämisestä paljon aikaa.

Komponenttien lisääminen MaterialUi-kirjastoa hyödyntäen oli suorastaan huumaavaa ja kirjastolle luotu dokumentaatio on yksi parhaista, mitä olen nähnyt.

Taka-alalle jätetty mobiilinäkymän luominen viimeisteltäessä sovellusta oli odotettua helpompaa ja tästä on kiittäminen komponenttien oletusasetuksia. Vaikka kirjaston päivitys helpotti monien komponenttien käyttämistä, hinta kaikelle tälle oli mahdollisten ongelmien ratkominen. Kehittäjänäkymässä saadut komponenttien ydintiedot olivat monen kerroksen alla ja se hidasti paikallistamista huomattavasti, johtaen usein komponenttien vaihtamiseen. Kirjaston uudistunut teemanhallintaa oli erittäin miellyttävä käyttää ja muutosten tekeminen on niin yksinkertaista, että siihen voi palata koska vain, jos ulkonäkö alkaa tökkimään. Uusi UseMemo-koukku mahdollisti myös todella sulavan vaihdon valoisan ja tumman teeman välillä ja tämän päivityksen tehokas toimiminen mobiililaiteissa oli positiivinen yllätys. Tilanhallintaan käytetty Redux teki sen mitä lupasi, mutta aikaisemmista kokemuksista ei tämän projektin kohdalla ollut hyötyä. Alkuperäisessä ideassa oli hallittavia tiloja vähemmän, koska ajatuksena oli kaikkien noodien tilanseuraamisen riittävyys. Tämä ei kuitenkaan riittänyt, vaan tilanhallinta päädyttiin aloittamaan alusta ja painottamaan hallinta näkyvissä olevan noodin ympärille, jättäen muut noodit toissijaiseksi. Muutoksien tekeminen kesti kauan, mutta pitäytymällä alkuperäisessä suunnitelmassa sovelluksen lopputulos olisi varmasti ollut saavutettua alkeellisempaa. Tilanhallintaa toteuttaessa oli myös todettava sen käytännöllisyys kyseisessä projektissa, jossa päivitykset ovat tiheitä ja tilan siirtäminen alemmille komponenteille olisi ollut huomattavasti vaikeampaa ilman globaalia tilaa. Toinen tilanhallinasta vahvistunut näkemys on sen käyttöönottoon vaadittu työ, jolloin tämänkaltaisen hallinnan toteuttaminen pienemmissä töissä on kyseenalaista.

Sain monia hyviä ideoita mahdollisesta opinnäytetyöstä, mutta koen että tämän projektin toteuttaminen on kehittänyt minua enemmän kuin muut vaihtoehdot olisivat kehittäneet. Projektissa käytetty geneerisyys, listojen manipulaatio tuntemattomilla avaimilla ja arvoilla antoivat jatkuvaa haastetta kehitystyölle. Jatkuvien haasteiden ansiosta, myös jokainen onnistumisen askel tuotti suurta mielihyvää ja antoi intoa jatkaa eteenpäin projektin parissa. Tieto siitä, että projektin onnistunut lopputulos tulisi oikeaan käyttöön ja olisi mahdollisesti myös tuotteistettavissa, pakottivat luomaan oikeasti hyvän lopputuloksen. Tehtävässä saadut opit ovat varmasti hyödynnettävissä uusissa projekteissa jo pelkästään geneerisyyden

takia, mutta koen myös, että työni jälki oli hyvää ja sitä pitää kopioida käyttöön myös tulevaisuudessa tarpeen vaatiessa. Ainoa sääntö, jonka jätin sovellukseen oli noodin parametrien avainten muokkaamisen mahdottomuuden. Loin mielessäni tapoja toteuttaa muokkaus niin, että avaimiakin olisi kyetty muokkaamaan, mutta sellainen toteutus olisi vaatinut todella suuren määrän yhdistelmien tutkimista ja tämä olisi varmasti hidastanut selainta, jos toteutus olisi ollut keskivertoa. Vaikka mahdollisuus muokata avaimia olisi ollut haastava toteuttaa, se ei ole täysin mahdotonta ja tulen varmasti palaamaan kyseisessä tulevaisuudessa.

Jatkossa haluan myös luoda näkymän noodien muodostamalle puulle, joka auttaisi käyttäjää havainnoimaan paremmin omaa rakennettaan. Sivupalkkien jälkeläisten järjestys on paikoitellen pielessä johtuen mahdollisuudesta luoda noodeja vaihtelevin tasoin. Ongelman korjaamiselle löysin vain yhden toteutustavan ja tämä vaatisi tietokannan muokkaamista niin, että se tallentaisi jälkeläisten syvyydet ja näiden, sekä avainten arvolla muodostettaisiin selkeät tasot jälkeläisille. Muilta osin, ja varsinkin palvelinpuolella toteutuneet ratkaisut ovat hiottuja, jäi vain halu oppia Djangon tarjoamaa Rest-kehystä vielä paremmin. Tulevaisuudessa haluan tutustua myös selainpuolen optioimintiin paremmin, sillä paikoitellen koin piirtymistahdin liian lujana. Edellä mainittujen lisäksi tulen varmasti löytämään parannuskohteita sovelluksesta, sen tuomien onnistumisien ansiosta ja koen tehdyn sovelluksen olevan yksi parhaistani ja selkeä näyte omasta kehittymisestäni alalla.

Lähteet

Antonios, M., Konstantinos, T., Spiliopoulos, G., Dimitrios, Z. & Anagnostopoulos, D. (13.4.2020). *MongoDB Vs PostgreSQL: A comparative study on performance aspects.* https://link.springer.com/content/pdf/10.1007/s10707-020-00407-w.pdf

Boyer, J. (17.6.2018). *How Secure Are Popular Web Frameworks? Here Is a Comparison*. https://www.veracode.com/secure-development/popular-web-frameworks

Cardelli, L. (3.1985). *BASIC POLYMORPHIC TYPECHECKING*. https://www.lucacardelli.name.com/Papers/BasicTypechecking.pdf

Enge, E. (23.3.2021). *Mobile vs. Desktop Usage in 2020*. https://www.perficient.com/insights/research-hub/mobile-vs-desktop-usage

Hirschauer, J. (20.9.2021). *What Is a CI/CD Platform and Why Should I Care?* https://harness.io/blog/what-is-cicd-platform-why-should-i-care/

Makai, M. (8.2017). *Object-relational Mappers (ORMs).* https://www.fullstackpython.com/object-relational-mappers-orms.html

Microsoft. (10.1.2021). *MATCH (Transact-SQL)*. https://docs.microsoft.com/en-us/sql/t-sql/queries/match-sql-graph

Musch, M. & Johns, M. (11.8.2021). *U Can't Debug This: Detecting JavaScript Anti-Debugging Techniques in the Wild*. USENIX Security Symposium. https://www.usenix.org/system/files/sec21-musch.pdf

Pandit, N. (10.2.2021). *What And Why React.js.* https://www.c-sharpcorner.com/article/what-and-why-reactjs/

Puntambekar, A. (2020). *Data Structures.* https://books.google.fi/books?id=9s8cEAAAQBAJ

Python Software Foundation. (11.2021). *General Python FAQ*. https://docs.python.org/3/faq/general.html#why-is-it-called-python

Python Software Foundation. (2021). *What is Python? Executive Summary*. https://www.python.org/doc/essays/blurb/

Reenskaug, T. (20.8.2003). *The Model-View-Controller (MVC)*. Oslo, Norja. http://heim.ifi.uio.no/trygver/2003/javazone-jaoo/MVC_pattern.pdf

Saikkonen, R. & Soisalon-Soininen, E. (2008). *Cache-sensitive Memory Layout.* Helsinki. https://www.springer.com/Cache-sensitive-Memory-Layout-for-Binary-Trees

Schaefer, L. (n.d). *What is NoSQL?* https://www.mongodb.com/nosql-explained

Skip, M. (24.2.2012). *Why is Python a dynamic language and also a strongly typed language*. https://wiki.python.org/moin/Why-is-Python-Dynamic

Stepanov, A. & Stroustrup, B. (1989). *Generic programming.*

https://dbpedia.org/page/Generic_programming

Thorben, J. (23.11.2017). *OOP Concept for Beginners: What is Abstraction?*

https://stackify.com/oop-concept-abstraction/

Venners, B. (13.1.2003). *The Making of Python. A Conversation with Guido van Rossum, Part I*. https://www.artima.com/articles/the-making-of-python

Weinberger, C. (14.2.2018). *NoSQL Performance Benchmark 2018 – MongoDB, PostgreSQL, OrientDB, Neo4j and ArangoDB*. https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018

Liite 1: Aktiivisen Noodin tilanhallinta

```
import nodeServices from "../services/nodes";

import { setNotification } from "./notificationReducer";

let config = {

  headers: {

    "Content-Type": "application/json"

  }

};

const activeReducer = (state = { node: null, parents: null }, action) => {

  if (action.error) {

    setNotification(action.error, 1000);

  }

  switch (action.type) {

    case "INIT_ACTIVE":

      return {

        ...state,

        node: action.content

      };

    case "UPDATE_ACTIVE_NODE":

      return action.content;

    case "CLEAR_ACTIVE":
```

Liite 2: Aktiivisen noodin tilanhallinta

```
      return { node: null, parents: null };
    default: return state;
  }
};
export const initializeActive = (token, id) => {
  return async (dispatch) => {
    const onSuccess = (node) => {
      dispatch({
        type: "INIT_ACTIVE",
        content: node
      });
    };
    const onError = (error) => {
      dispatch(setNotification(error, 1200));
    };
    try {
      if (token) {
        config.headers["Authorization"] = `Token ${token}`;
        const node = await nodeServices.getOne(id, config);
```

Liite 3: Aktiivisen noodin tilanhallinta

```
            return onSuccess(node);
      }
    } catch (error) {
      return onError(error);
    }
  };
};
export const updateActive = (id) => {
  return async (dispatch) => {
    const onSuccess = (node, parents) => {
      dispatch({
        type: "UPDATE_ACTIVE_NODE",
        content: { node, parents }
      });
    };
    const onError = (error) => {
      dispatch(setNotification(`${error} occured`, 1200));
    };
    try {
      const node = await nodeServices.getOne(id, config);
```

Liite 4: Aktiivisen noodin tilanhallinta

```
const parents = await nodeServices.getParents(id, config);

      return onSuccess(node, parents);

    } catch (error) {

      return onError(error);

    }

  };

};

export const clearActive = () => {

  return async (dispatch) => {

    dispatch({

      type: "CLEAR_ACTIVE"

    });

  };

};

export default activeReducer;
```

Liite 5: Optimoitu noodinäkymä

```
const ContainerContent = () => {
  const dispatch = useDispatch();
  const nodes = useSelector((state) => state.nodes);
  const user = useSelector((state) => state.user);
  const activeNode = useSelector((state) => state.active.node);
  useEffect(() => {
    // When rendering first time try to get user
    if (window.localStorage.getItem("loggedUser") && !user) {
      dispatch(setNotification("Retrieving User", 800));
      dispatch(getActiveUser(window.localStorage.getItem("loggedUser")));
    }
  }, []); // eslint-disable-line react-hooks/exhaustive-deps
  useEffect(() => {
    // If user is found and nodes is null try to find nodes
    if (user && !nodes) {
      dispatch(setNotification("Finding Nodes", 800));
      dispatch(initializeNodes(window.localStorage.getItem("loggedUser")));
    }
  }, [user, nodes]);
```

Liite 6: Optimoitu noodinäkymä

```
 useEffect(() => {
    // Just once if nodes are loaded and the user has one node init active
    if (!activeNode && nodes && nodes.length > 0 ) {
      let mother = nodes.find((node) => (node.parent = "null"));
      if (mother) {
        dispatch(
          initializeActive(
            window.localStorage.getItem("loggedUser"),
            mother.node_id
          )
        );
      }
    }
  }, [nodes]);
  if (!user) {
    return (<ContainerLogin />)
}
  if (nodes === null) return <></>;
  return (
```

Liite 7: Optimoitu noodinäkymä

```
 <Container maxWidth='xl'>
 <NavBar />
      <Grid container spacing={2}>
        <Grid item xs={12} sm={2}>
          <SideBar />
        </Grid>
        <Grid item xs={12} sm={10}>
          <Card>
            <CardContent>
              {nodes.length === 0 ? <NewProjectDialog /> : <NodeView />}
            </CardContent>
          </Card>
        </Grid>
      </Grid>
    </Container>
  );
};
export default ContainerContent;
```

Liite 8: Käytetyt mallit

```
# Node Model

class Node(models.Model):

    node_id = models.AutoField(primary_key=True)

    name = models.CharField(default="no name given",max_length=255)

    sort_order = models.IntegerField(default=10)

    # Node has only one parent

    parent =
models.ForeignKey("nodeller_backend.Node",related_name="children",null= True,
blank= True, on_delete=models.CASCADE)

    # Node can have 0 , 1 or many children

    descendants = models.ManyToManyField("nodeller_backend.Node",
related_name="ancestors", through="nodeller_backend.Closure")

    data = JSONField(db_index=True,default = dict)

    owner = models.ForeignKey(User,related_name="nodes",on_delete=models.CASCADE,
null=True)

    def __str__(self):

        return self.name

    class Meta:

        app_label = "nodeller_backend"

        ordering = ["sort_order"]
```

Liite 9: Käytetyt mallit

```
# Sql View's model in django

class Closure(models.Model):

    path = ArrayField(base_field=models.IntegerField(), primary_key = True)

    ancestor = models.ForeignKey("nodeller_backend.Node",
related_name="+",on_delete=models.CASCADE)

    descendant = models.ForeignKey("nodeller_backend.Node",
related_name="+",on_delete=models.CASCADE)

    depth = models.IntegerField()



    class Meta:

        app_label = "nodeller_backend"

        # This creates a pseudotable

        managed = False
```